Grant Agreement No.: 101070473
Call: HORIZON-CL4-2021-DATA-01
Topic: HORIZON-CL4-2021-DATA-01-05
Type of action: HORIZON-RIA

D2.1 Scenarios, Requirements and Reference Architecture – v.1

Revision: v.1.0

Work package	WP 2
Task	Tasks T2.1, T2.2, T2.3
Due date	28/02/2023
Submission date	30/04/2023
Deliverable lead	POLITO
Version	1.0
Authors	Fulvio Risso, Marco Iorio, Stefano Galantino, Jacopo Marino, Fulvio Valenza, Riccardo Sisto (POLITO) Alessandro Cannarella (TOP-IX) Vlad Coroama, Nasir Asadov (TUB) Stefano Braghin, Ambrish Rawat, Mohamed Suliman (IBM)

Funded by Horizon Europe
Framework Programme of the European Union

	Andy Edmonds (TER) Antonio Skarmeta, Alejandro M. Zarca, José Manuel Bernabé, José Francisco Pérez (UMU) Domenico SIracusa (FBK) Emna Amry (CYSEC) George Kornaros (UMU) Marcello Coppola (STM) Elisa Albanese, Roberta Terruggia (RSE) Eduard Marin (TID)
Reviewers	Silvio Cretti (FBK) Amjad Majid (MAR)

Abstract	This document provides the general architectural framework defined in FLUIDOS for the creation of a computing continuum, spanning across multiple technological domains (e.g., cloud, edge, single devices, embedded devices) and multiple administrative boundaries (e.g., enterprise campus, telco operator, cloud provider).
Keywords	Computing continuum; liquid computing; FLUIDOS architecture

Document Revision History

Version	Date	Description of change	List of contributor(s)
V0.1	15/01/2023	1st version of the template for comments	Margherita Facca (MAR)
V1.0	30/04/2023	Final version v1.0	Fulvio Risso (POLITO)

Disclaimer

Copyright notice

Project co-funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	R	
Dissemination Level		
PU	Public, fully open, e.g., web	
SEN	Sensitive, limited under the conditions of the Grant Agreement	
Classified R-UE/ EU-R	EU RESTRICTED under the Commission Decision No2015/ 444	
Classified C-UE/ EU-C	EU CONFIDENTIAL under the Commission Decision No2015/ 444	
Classified S-UE/ EU-S	EU SECRET under the Commission Decision No2015/ 444	

** R: Document, report (excluding the periodic and final reports)*

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.

Funded by Horizon Europe
Framework Programme of the European Union

EXECUTIVE SUMMARY

This document provides a general architectural framework for the creation of a computing continuum, spanning across multiple technological domains (e.g., cloud, edge, single devices, embedded devices) and multiple administrative boundaries (e.g., enterprise campus, telco operator, cloud provider).

We include here an exhaustive analysis of the state-of-the-art with respect to the different topics covered by this project (e.g., single node architecture, computing continuum, security, energy), spanning across both research results and suitable technological tools. This would provide a common and solid ground for the FLUIDOS architecture, both in terms of research perspective and technological feasibility.

The FLUIDOS architecture is presented mostly in a technology-agnostic fashion. All technology-dependent and implementation related details are addressed in deliverables provided by the following work-package: WP3 (FLUIDOS node), WP4 (Intent-based computing continuum), WP5 (Security) and WP6 (Energy and cost-effective infrastructure).

Funded by Horizon Europe Framework Programme of the European Union

TABLE OF CONTENTS

Funded by Horizon Europe Framework Programme of the European Union

Funded by Horizon Europe Framework Programme of the European Union

Funded by Horizon Europe Framework Programme of the European Union

Funded by Horizon Europe
Framework Programme of the European Union

LIST OF FIGURES

Funded by Horizon Europe Framework Programme of the European Union

Funded by Horizon Europe Framework Programme of the European Union

LIST OF TABLES

Funded by Horizon Europe
Framework Programme of the European Union

ABBREVIATIONS

IP	Internet Protocol
TCP	Transmission Control Protocol

Funded by Horizon Europe Framework Programme of the European Union

1 INTRODUCTION

In the last years, containerization has increasingly gained popularity as a lightweight solution to package applications in an interoperable format [1], independently of the target infrastructure. This uniform substratum paved the way for the cloud native revolution, with novel applications shifting their focus from single servers to entire data centers, and where dedicated orchestrators manage the lifecycle of microservice applications. As of today, Kubernetes emerged as the de-facto open-source framework for container orchestration, bridging the semantic gaps across competing infrastructure providers [2]. With the rise of the edge and fog computing paradigms [3][4][5] as solutions accounting for geographical closeness, reduced latency and improved privacy, the same approaches are being progressively extended towards smaller data centers at the network border, benefiting from uniform primitives to foster service agility.

Despite the emergence of common interfaces for applications orchestration being key towards a real edge to cloud continuum [6][7], industry-standard approaches handle each infrastructure as a **multitude of (connected) isolated silos instead of a unique virtual space**. This leads to a **sub-optimal fragmented view of the overall available resources, preventing the seamless deployment of fully distributed applications**. Indeed, edge data centers cannot depend on a single centralized control plane, for resiliency (i.e., preventing failure propagation in case of network partitioning) and performance reasons, as orchestration platforms typically suffer if nodes are geographically spread over high-latency WANs [8][9][10]. Besides the edge landscape, resource fragmentation also affects larger data centers, with many companies increasingly witnessing the cluster sprawl phenomenon [11][12]. This trend finds its roots in scalability concerns, in the hybrid-cloud (i.e., the combination of on-premises and public cloud) and multi-cloud approaches [13], which aim for high availability, geographical distribution and cost-effectiveness, while granting access to the breadth of capabilities offered by competing cloud providers. Additionally, non-technical requirements such as law regulations, mergers and acquisitions, physical isolation policies and separation of concerns contribute to the proliferation of clusters.

Fragmentation also hinders the potential dynamism in the workload placement [14][15][16], forcing each application to be assigned upfront to a specific infrastructure. No resource compensation is ever possible, hence preventing jobs from transparently moving from an overloaded cluster, e.g., due to unexpected spikes of requests, to another one, underused and potentially offering better performance. At the same time, the deployment of complex applications composed of multiple microservices, each one with specific requirements (e.g., low latency, high computational power, access to specialized hardware, ...), as well as the enforcement of proper geographical distribution and high-availability policies, requires the interaction with different infrastructures. However, this prevents relying on the single point of control, which would coordinate the deployment of arbitrarily complex applications across the entire resource continuum, no matter how many nodes and clusters it is composed of.

Funded by Horizon Europe Framework Programme of the European Union

Accounting for these demands, this project advocates the opportunity for a novel architectural paradigm, called **liquid computing**[1] in the seminar paper [82], which builds upon and extends the well-established cloud and edge computing approaches towards an endless computing continuum. Overall, the resulting computing domain abstracts away the specificity of each cluster, presenting to the final users, either actively participating as actors or simply renting off-the-shelf services, a unique and borderless pool of available resources, the so-called big cluster. Thanks to this abstraction, applications are no longer constrained in a specific silo, but free to fly in the entire infrastructure, selecting the most appropriate location depending on its requirements (e.g., a user facing service may be replicated at the edge to account for low latency, while another might be constrained to European infrastructures to comply with GDPR), and the available resources, while retaining full compatibility (hence, models, tools, and commands) with vanilla Kubernetes.

1.1 The FLUIDOS Computing Continuum

At a first sight, the computing continuum seems to be a reality today, without the necessity of big investments in terms of technology and research. For instance, many software applications already rely on multiple components which are installed and operated in different locations (e.g., data gathering at the far edge, data aggregation at the telco edge, deep data processing in the cloud), hence apparently already implementing a computing continuum.

This section highlights how the FLUIDOS approach to Computing Continuum (a.k.a., *liquid computing*) has three distinctive characteristics that are not matched by existing approaches.

1.1.1 Deployment transparency

When an application composed of multiple microservices such as in of Figure 1 is deployed, each component must be explicitly configured to land on a specific target, e.g., edge datacenter vs. cloud. The location of each component is therefore fixed and decided a-priori; no modifications are allowed with respect to the location of each different component, unless starting a new re-deployment phase. Consequently, any possible dynamic optimization that could be carried out at run-time is more complex, as it requires the presence of an overarching orchestrator that re-deploys all the required components in the new optimal location, which is something that does not exist with the current technology.

[1] This term was first coined in 2014 by InfoWorld (G. Gruman, "Welcome to the next tech revolution: Liquid computing," https://www.infoworld.com/article/2608440/article.html, Jul. 2014) as a synonym of pervasive computing, i.e., the capability of keep working on a given task across multiple devices such as PCs and tablets. This paper refers to a broader concept, which encompasses the creation of a resource continuum composed of cloud and edge infrastructures, on-premises clusters, as well as, in its widest form, single end-user and IoT devices.

Funded by Horizon Europe Framework Programme of the European Union

Instead, the **FLUIDOS intent-based interface guarantees that each micro-service is started in the best location and provides also dynamic optimizations if required**. Hence, DevOps have simplified operations with FLUIDOS, as all services leverage a single point of deployment and control, while the FLUIDOS "magic" will start the above services in the most appropriate location given the service requirements and the infrastructure status.

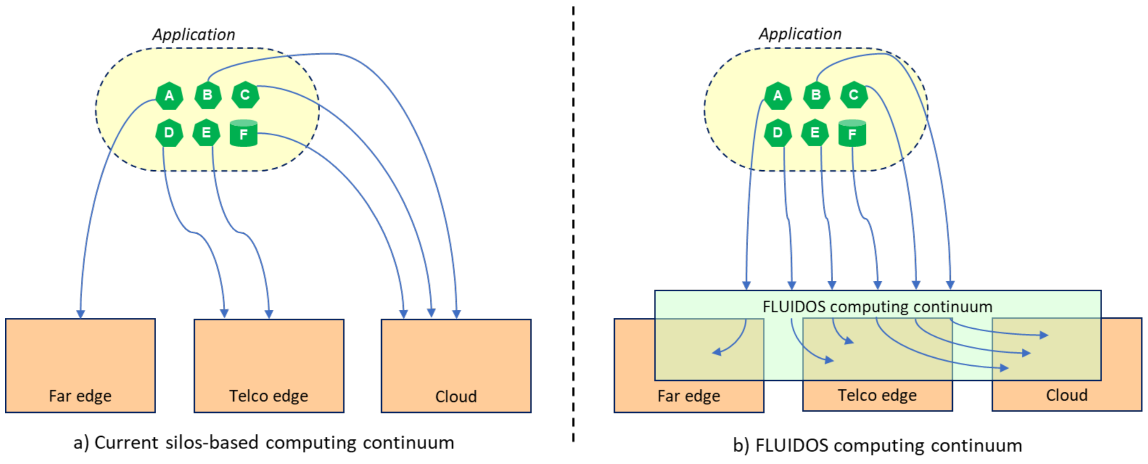

Application
A
B
C
D
E
F
Far edge
Telco edge
Cloud
a) Current silos-based computing continuum
Application
A
B
C
D
E
F
FLUIDOS computing continuum
Far edge
Telco edge
Cloud
b) FLUIDOS computing continuum

Figure 1. Computing continuum: deploying apps with traditional approach vs. FLUIDOS.

1.1.2 Communication transparency

The communication between different microservices is different whether they belong to the same communication space (e.g., the same Kubernetes cluster) or not. For instance, by default all the communications inside a cluster are allowed, while all the communications from external components are forbidden. Furthermore, a service that accepts communications from inside the cluster requires primitives (e.g., the Kubernetes ClusterIP service) that are different from the ones used to accept data from outside the cluster (e.g., the Kubernetes NodePort or LoadBalancer services). Therefore, services must be explicitly configured to talk to each other based on their respective location, hence further complicating the actions required to carry out a possible re-deployment of the components mentioned above. It is worth mentioning that some technologies can partially overcome this problem, e.g., when the communication among micro-services occurs through message brokers (e.g., technologies based on publish/subscribe primitives, such as Kafka). However, this requires all the micro-services to rely on the above communication primitives, which is not always possible because of the intrinsic characteristics of the pub/sub technology (e.g., cannot guarantee reduced latency), or because applications do not use this technology (e.g., they are based on HTTP or gRPC protocols).

The FLUIDOS approach to the computing continuum is visible in the right part of the figure below: a virtual cluster spanning across multiple real clusters is envisioned, and applications are operating on the above virtual space. **With FLUIDOS, all communications between micro-services are mediated by the FLUIDOS virtual network fabric, which guarantees seamless**

Funded by Horizon Europe Framework Programme of the European Union

communications independently from the location of each microservice. Hence, the communication between two services belonging to the same virtual space happens as they were inside the same cluster, hence avoiding the necessity of complex and error-prone configurations that must capture the fact that services that in the same cluster, or deployed in two different clusters.

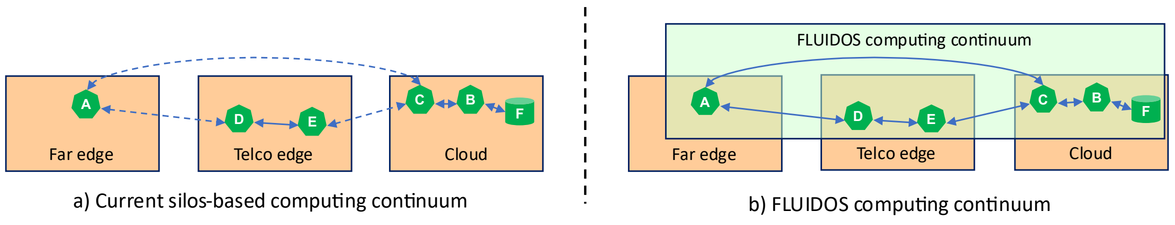

Far edge
Telco edge
Cloud
A
B
C
D
E
F
a) Current silos-based computing continuum
FLUIDOS computing continuum
b) FLUIDOS computing continuum

Figure 2. Computing continuum: service-to-service communications with traditional approach vs. FLUIDOS.

1.1.3 Resource availability transparency

With the current technology, each microservice can use only the resources that are inside its own cluster. This statement holds for both normal operations (e.g., when the service is started), and when an update (e.g., automatic scaling) is requested. This behavior prevents a service from using available resources that located in other parts of the continuum, hence possibly ending up in a service disruption (e.g., due to a lack of resources) even in presence of available resource, which cannot be leveraged because they are located in another part of the continuum.

This problem is not very important in cloud datacenters, where it is very unlikely to run into a lack of resources. Even in the case of small clusters obtained by acquiring only a small subset of the datacenter resources, the problem is irrelevant thanks to the capability to (automatically) resize the cluster, e.g., by adding/removing worker nodes to the given cluster. Instead, this problem is more important when considering a small pool of resources available at the edge, in which usually a few servers are available. The capability to acquire new physical resources, when needed, needs to be achieved by leveraging other worker nodes possibly available in the closest vicinity.

FLUIDOS overcomes the above limitation by enabling the creation of a virtual computing space spanning across multiple physical domains, hence enabling a service (which has been started in the virtual space) to leverage all the resources belonging to the same virtual domain, independently from their physical location. Hence, in FLUIDOS a service can seamlessly scale based upon the availability of resources within the entire virtual infrastructure, e.g., ending up having one instance running in the telco edge, and another in the cloud datacenter, hence blurring the current rigid cluster boundaries.

Funded by Horizon Europe Framework Programme of the European Union

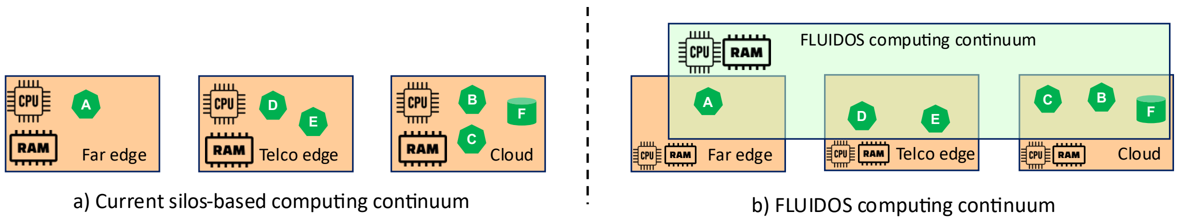

CPU
RAM
A
Far edge
D
E
Telco edge
B
F
C
Cloud
a) Current silos-based computing continuum
FLUIDOS computing continuum
b) FLUIDOS computing continuum

Figure 3. Computing continuum: available resources with traditional approach vs. FLUIDOS.

1.2 Liquid Computing vs. Current Computing Continuum

Given the above consideration, **the computing continuum envisioned by FLUIDOS** (a.k.a., Liquid Computing) is not simply the capability to deploy services in multiple sites datacenter clusters or devices. It **defines a virtual space**, spanning across multiple technological domains (e.g., one cluster on-prem, another running on the cloud; or one single device, and a cluster running on edge servers; etc.) and, potentially, across multiple administrative boundaries (e.g., two mobile edge operators sharing their resources), **in which the three properties presented above hold: Deployment transparency, Communication transparency, and Resource availability transparency**, with the advantages presented above.

Obviously, the creation and the maintenance of the liquid continuum has a cost, e.g., in terms of computing (additional software services running on the nodes) and communication overhead (exchanging messages to synchronize data in the different entities). **FLUIDOS has the ambition to determine the best trade-off between the advantages brought by the liquid computing** (e.g., in terms of better utilization of available resources) **and the corresponding costs, which could be used to determine where to put the boundaries of the computing continuum, e.g., at the edge of the infrastructure**.

1.3 The technological substrate: Kubernetes

FLUIDOS starts with a strong (and potentially controversial) assumption: the chosen reference technology is Kubernetes. This stems from several considerations:

- **Cloud-native**: Kubernetes is considered the most promising platform to provide cloud-native services, which provide unprecedented agility and efficiency compared to the previous world of VMs.
- **Scalability**: Kubernetes is designed to scale applications across many servers in a cluster, allowing them to handle large traffic loads without downtime.

Funded by Horizon Europe Framework Programme of the European Union

- **Portability**: Kubernetes is platform-agnostic, meaning it can be used on-premises or in the cloud, and supports multiple cloud providers, such as AWS, Google Cloud Platform, and Azure.
- **Large-to-Small scale support**: multiple flavors of Kubernetes exist, with support for both large deployments (e.g., cloud datacenter), and small-scale deployments (e.g., even resource-constrained individual devices), hence becoming the natural candidate to build the META-OS concept upon.
- **Flexibility**: Kubernetes provides a range of features to manage containerized applications, such as scaling, deployment, updates, and monitoring. It also supports various container runtimes, including Docker and CRI-O.
- **Resource Optimization**: Kubernetes can optimize resource utilization by automatically allocating containers based on available resources and workload demands.
- **Community Support**: Kubernetes has a large and active open-source community that contributes to its development and maintains a vast ecosystem of add-ons and tools.

Nevertheless, although the FLUIDOS project assumes Kubernetes at its technological foundation, the overall architecture and most of the choices and proof-of-concept components developed in this project aim at having a more general breadth, hence potentially enabling their reuse with other technological substrates.

Funded by Horizon Europe Framework Programme of the European Union

2 THE FLUIDOS VISION

We envision liquid computing as a continuum of resources and services allowing the seamless and efficient deployment of applications, independently of the underlying infrastructure. We present here the main characteristics of liquid computing, followed by the most significant deployment scenarios.

2.1 MAIN CHARACTERISTICS

We believe this paradigm shall be composed of the following four distinguishing characteristics.

2.1.1 Intent-driven

A consumer can assign to each workload the desired execution constraints through high-level policies, without knowing about the infrastructural details. Overall, liquid computing brings the cattle service model [17] to a greater scale. Like servers in a data center, with no one caring about where each task is executed, if requirements are fulfilled, this paradigm blurs the cluster borders so that users are relieved from selecting a specific infrastructure for their applications. Yet, different clusters are associated with different properties (e.g., in terms of geographical location and security characteristics) and, indeed, this is one of the main driving reasons behind cluster sprawling. Thus, it is of utmost importance the adoption of an intent-driven approach, allowing final users to enrich each workload with a set of high-level policies to express the associated constraints (e.g., geographical locality and spreading, costs, capabilities, ...); automated schedulers shall select the best execution place across the entire border-less infrastructure, depending on the available resources and enforcing in concert the user-specified policies. Yet, we deem at the same time the resource continuum abstraction to enable more contextualized scheduling decisions (given the knowledge about the entire infrastructure), allowing for further optimizations and better scalability compared to the siloed approach.

2.1.2 Decentralized architecture

The resource continuum stems from a peer-to-peer approach, with no central point of control and management entities, as well as no intrinsically privileged members. Following a decentralized and peer-based model like the Internet, the liquid computing approach fosters the coexistence of multiple actors, including larger cloud providers, smaller, territory-linked enterprises, and even small office/homeowners. Indeed, each entity can autonomously and dynamically decide who to peer with (hence, share resources), similarly to the concept of Autonomous Systems in the Internet inter-domain routing. A dynamic discovery and peering

Funded by Horizon Europe Framework Programme of the European Union

protocol is in charge of the automatic identification of available peers and the negotiation of peering contracts based on the demands and offers of each actor, along with their specific constraints; optionally, the above operations could also be delegated to an intermediate dedicated entity such as a broker. No sensitive information disclosure is mandated (e.g., infrastructural setup), with the entire process possibly involving only the request for a certain amount of abstract resources (e.g., CPU, memory, ...) and the offer of available ones, together with the associated cost. Besides peering establishment, decentralization also concerns the preserved ability of each cluster to evolve independently, thanks to the local orchestration logic, and the support for the creation of arbitrary topologies, with different points of entry for the deployment of different workloads.

2.1.3 Multi-ownership

Each actor maintains full control of his own infrastructure while deciding at any time how many resources and services to share and with whom. Although single clusters are expected to be under the control of a single entity, the entire resource ocean would likely span across different administrative domains. Once a new peering is established, the control plane of the target infrastructure is in charge of configuring the appropriate isolation primitives (e.g., resource quota, network and security policies, ...), based on the underlying orchestration capabilities, to enforce the shared resource slice and prevent noisy neighbors' phenomena. Specifically, we foresee a shared security responsibility model, with the provider responsible for the creation of well-defined sandboxes and the possible provisioning of additional security mechanisms (e.g., secure storage) negotiated at peering time. Requesters, on the other hand, are expected to take measures to fortify their applications (similarly to public cloud computing) and to configure for each sensitive component the appropriate policies to ensure it is scheduled on security-compliant infrastructures only (e.g., private data is processed locally).

2.1.4 Fluid topology

Members can join and leave the virtual continuum at any time, independently from their infrastructure size, from enterprise-grade data centers to IoT and personal devices. Generalizing traditional federation approaches, liquid computing aims at supporting highly dynamic scenarios, with frequent and unexpected (or, in other scenarios, explicitly desired) connections and disconnections. Besides spanning across public and private data centers, as well as edge clusters, the resource continuum encompasses also single devices. This would include IoT, industrial and domestic scenarios, all characterized by a multitude of independent appliances typically dedicated to specific tasks (e.g., machine tools control, home automation, monitoring, ...), which could greatly benefit from this paradigm, transparently leveraging the shared resources to offload computations and extend their capabilities [15].

Funded by Horizon Europe Framework Programme of the European Union

2.2 DEPLOYMENT SCENARIOS

We foresee three high-level deployment scenarios to be mostly enabled and fostered by liquid computing (Figure 4).

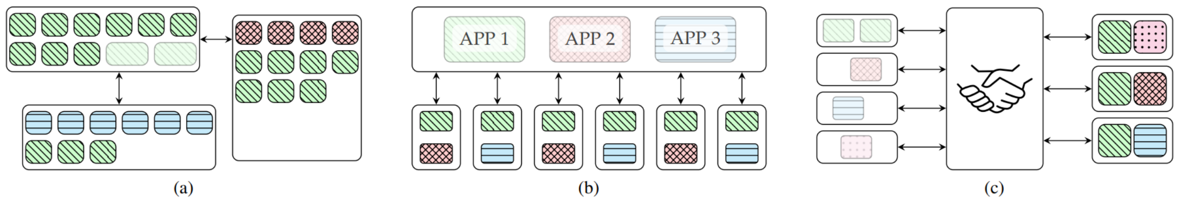


Figure 4. Three main deployment scenarios for Liquid computing.

2.2.1 Elastic cluster

The liquid computing paradigm reduces the fragmentation of scattered clusters thanks to the possibility of transparently leveraging the resources available in other locations, which enables it to balance and absorb load spikes (cloud bursting). This applies to edge-computing scenarios, whose pervasiveness is typically achieved at the expense of computational capacity but is visible also in more traditional cloud contexts where multiple clusters are active. For example, elastic clusters can be used to support multi-cloud strategies (no single vendor lock-in) and geographically distributed multi-cluster deployments (e.g., due to cost optimization, latency, redundancy, or legislative concerns). Resource sharing can involve both on-premise and public data centers, hence deploying latency-sensitive applications close to the final users, while benefiting from the virtually infinite computational capacity featured by larger infrastructures for resource-intensive tasks. Thanks to the decentralized approach and the support for peering contracts, resource sharing can occur also when clusters are under the control of different organizations, i.e., they belong to different administrative domains.

2.2.2 Super cluster

We are currently witnessing scenarios in which a company owns hundreds, or even thousands, of small clusters, often located at the edge of the network, such as a large telecom operator. In this context, the liquid computing paradigm enables a higher-level, super cluster abstraction, representing the single point of entry that can transparently deploy and control applications hosted by the entire infrastructure. Although apparently centralized, each level-2 cluster maintains its own local orchestration logic, hence being resilient to network outages and preventing possible synchronization issues arising with relatively high latency WAN links [8][9][10]. In addition, thanks to the super cluster, the administrative burden is greatly reduced, enabling a borderless orchestration that geographically distributes and controls the tasks from a single point of entry, without the necessity of an explicit interaction with the underlying clusters. This scenario facilitates the automatic migration of applications from one cluster to another, which helps when dealing with disaster recovery, infrastructure interventions, scaling,

Funded by Horizon Europe Framework Programme of the European Union

or placement optimization. In addition, it greatly simplifies the replication of jobs across clusters (hence management, monitoring and troubleshooting) as it leverages existing primitives that operate on nodes of the "super cluster"; for example, an operator can easily replicate the same service on a subset of its edge clusters to serve all the end users present in their close vicinity. Finally, this approach can be combined with the elastic cluster for increased dynamism, thus benefiting from the single point of entry for workloads replication, while enabling at the same time the offloading of "bursty" workloads from the edge to the cloud.

2.2.3 Brokering cluster

Complex applications require complex infrastructural setups, accounting for both resiliency and performance. While larger cloud providers could theoretically offer a sufficiently wide catalog of services to satisfy most demands, relying on a single vendor increases lock-in and potentially leads to cost inefficiencies. In this context, we believe that liquid computing could foster the creation of new Resource and Service Exchange Points (RXPs), with third party entities behaving as brokers between consumers and providers. Consumers would then only need to peer (both technologically and contractually) with a single RXP to immediately benefit from the entire set of resources (cloud and edge data centers, ...) and ready-to-use services therein offered. This would reduce the complexity and the operational costs, especially for smaller companies, lacking the bargaining power of larger enterprises. Resource providers, at the same time, would be encouraged to participate, to easily reach a wider turnout of interested customers. Additionally, even small actors, such as the ones operating at the edge of the network, would be enabled to participate in the edge-cloud market, possibly in a fair competition with far larger giants that may not have enough resources to serve a given geographical area, in a way that looks like the energy market in which millions of tiny producers are aggregated by larger buyers.

2.3 The "Liquid Computing" pillars

This section presents the main technical building blocks required to materialize the liquid computing vision, assuming Kubernetes as the orchestration platform leveraged by the underlying clusters. In fact, in our opinion, Kubernetes represents a key enabler for liquid computing, thanks to its capillary diffusion in data centers of any size [18], as well as the support for single devices and IoT computing by means of lightweight distributions, such as k3s [19]. To this end, the recent Microsoft's backed Akri project [20] goes even further, introducing an abstraction layer to dynamically interconnect to this platform the variety of sensors, controllers, and MCU class devices typically present at the very edge of the network. At the same time, Kubernetes can be easily extended through both custom APIs and logic, allowing to transparently integrate liquid-computing related aspects, as well as to semantically enrich the ecosystem and introduce new services that may be shared with peered clusters.

Funded by Horizon Europe Framework Programme of the European Union

Hence, Kubernetes can become the underlying platform the resource continuum is built upon, and it is conceptually similar to an overarching operating system. Still, the key concepts are more general and can be applied with no difference to other orchestrators, such as OpenStack, or even to a mix thereof.

2.3.1 Dynamic Discovery and Peering

The first key enabler is the discovery and peering function. It fosters the decentralized governance approach typical of a peer-to-peer architecture, preventing the need for central management entities and full administrative control over the entire infrastructure. Additionally, it is responsible for the liquid computing dynamism, allowing for new peering relationships to be established and revoked at any time, compared to the manual coordination required by static federation approaches. In this context, we define peering a unidirectional resource and service consumption relationship, with one party (i.e., the consumer) granted the capability to offload tasks and/or consume services in a remote cluster (i.e., the provider), but not vice versa. This allows for maximum flexibility in asymmetric setups, while transparently supporting bidirectional peerings through their combination.

Overall, this module deals with four main tasks. (i) *Discovery*, to identify candidate clusters to peer with, including large enterprise domains, as well as possibly local independent appliances (e.g., IoT devices). (ii) *Authentication*: given the list of feasible candidates obtained during the previous step, optionally filtered through user-configured criteria, it is responsible for the establishment of a secure communication channel with each selected counterpart. Still, resource offloading is not yet possible at this point, being the granted authorizations related to peering establishment steps only. (iii) *Resource negotiation*, involving the exchange of request and offer messages to identify the shortlist of clusters selected for resource offloading. The entire process is policy-driven, with decision modules local to each cluster determining at each step whether to proceed with the negotiation or to abort the process. As a representative example, an offering cluster might implement complex business logic to determine the appropriate prices based on current demands and available resources, accounting for resource brokering and reselling scenarios. Consumers, on the other hand, may filter and rank the received offers by means of appropriate criteria, possibly including compliance with the request constraints, cost, additional attributes, past experience, and more. The negotiation process culminates with the mutual agreement between a consumer and a provider. To this end, we envisage the adoption of smart contracts [21] to formalize the exchange in terms of money and resources, especially in the case of inter-administrative domain peerings. (iv) *Peering finalization*: once resource negotiation is completed, the peering relationship needs to be finalized, leading to the exchange of the preparatory parameters required for subsequent computation offloading, as well as the setup of isolation mechanisms and the granting of the suitable permissions in the target cluster.

Funded by Horizon Europe Framework Programme of the European Union

2.3.2 Hierarchical Resource Continuum

Once peering relationships have been established towards one or more targets, the local cluster gains logical access to remote resource slices. Yet, these need to be properly exposed for application offloading through a continuum abstraction, while respecting the limited knowledge propagation and the multi-ownership constraints. Moreover, we consider API transparency to be of utmost importance to foster its widespread adoption, thanks to the introduction of no disruption in well-established deployment and administration practices, as well as the immediate support for existing management solutions. Being traditional clusters composed of multiple nodes, each one mapping to a physical or virtual server, we propose to represent peered clusters through local, virtual, big nodes. Local, as attached to the consuming cluster; virtual, since they abstract a set of remote resources possibly unrelated from the underlying hardware; and big, being potentially much larger than classical nodes (in terms of available capabilities), as backed by an entire data center slice. The node concept perfectly complies with the requirement of sharing limited information, hence abstracting peered clusters only in terms of the aggregated resources currently being shared, with no additional details regarding its actual internal configuration. At the same time, it leads to overall better scalability, given the reduced amount of data synced among different clusters. This approach opens up for two possible cluster models. First, extended clusters, encompassing a combination of traditional physical Kubernetes nodes (i.e., workers), and virtual ones. This could be suitable for the resource optimization and RXP consumer use-cases, to allow borrowing external computational capacity to overcome local limitations. Second, virtual clusters, characterized by the absence of local workers. Combining only virtual nodes, they provide a single point of control abstraction to simplify the deployment of applications on user-defined slices of the underlying infrastructure. Moreover, they represent a key enabler for resource brokering, aggregating the resources offered by multiple providers (each one mapped to a virtual node) for reselling.

The virtual node abstraction leads the underlying orchestration platform (e.g., Kubernetes) to consider the above nodes as valid scheduling targets, hence allowing traditional workloads to be transparently assigned to remote clusters. No differences are perceived by the final users, who simply benefit from the larger number of available resources. This approach brings to a hierarchical representation of the resource continuum.

When a new workload is deployed in the local cluster, the scheduler first selects the optimal node for its execution. Then, if the target is a virtual node, the workload is remapped to the corresponding remote cluster, where a second scheduling round is started to identify the physical server where it will be executed upon. While considering a two-layer scheduling in this example, the approach can easily generalize to multiple levels if needed, depending on the number of virtual node redirections. Hence, allowing scheduling decisions to occur at different abstraction layers, reducing the overall number of feasible candidates to consider at each step and potentially increasing the resulting accuracy. Once more, compliance with standard Kubernetes APIs enables vanilla schedulers to deal out-of-the-box with extended

Funded by Horizon Europe Framework Programme of the European Union

clusters. However, custom scheduling logic might be appropriate in certain scenarios, allowing for further optimizations thanks to the knowledge about the semantics of the peering relationship (e.g., monetary costs, network characteristics, geographical distance, QoS). In both cases, end-users can easily enforce domain-specific constraints through Kubernetes standard high-level policies (i.e., selectors and affinities) to assign workloads to slices of nodes and ensure replicas spreading. Hence, sticking to an intent-driven approach, while requiring no modifications in standard application deployment workflows.

2.3.3 Resource and Service Reflection

Each virtual node is responsible for its allocated workloads, whose execution is delegated to the remote cluster. Hence, selected control plane information should be present both in the local cluster (required to fulfill the requirement of the virtual node abstraction) and in the remote cluster (enabling the remote-control plane to carry out its operations). This introduces the resource reflection concept: objects exist both in their native form (i.e., in the local cluster), and in their shadow form, remotely. Indeed, applications most likely require accessory artifacts for proper execution (e.g., configurations, authorization tokens, network endpoints, etc.), which then need to be reflected in the target cluster. The resource reflection logic enforces the transparent realignment between the two digital twins of the same artifact across the different domains, while ensuring the desired information opacity properties (i.e., omitting or masquerading data that should not be propagated) and resolving possible conflicts which may arise in the remote infrastructure (e.g., naming collisions, different underlying technologies, etc.). Overall, it shall support the propagation of both local modifications (e.g., the change in a user configuration) — outgoing reflection — and of remote status changes (e.g., an application is being restarted due to a crash), hence allowing for proper inspection — incoming reflection. Service endpoints represent one of the most important reflected information, enabling an application running on one cluster to be reachable (hence, consumable) from another cluster. This may require the close coordination of the network fabric to disambiguate and transparently translate possible overlapped network addresses used in the communication flows.

The clever reflection of the required information is the key to achieve objectives such as robustness, enabling clusters to evolve also in case of network disconnections, and scalability, reducing the amount of synced data.

2.3.4 Network Continuum

According to the virtual nodes approach, different components of the same application may be spread across multiple clusters. Still, the resource continuum, alone, is not sufficient to ensure their correct execution, as the various microservices most likely need to interact among each other. Orchestration platforms typically implement internal communication by means of private IP addresses, resorting to public ones only for user-facing services. Hence, they are unsuitable for direct (pod-to-pod) cross-cluster interactions and require the introduction of an

Funded by Horizon Europe Framework Programme of the European Union

appropriate network fabric responsible for the transparent communication between microservices, no matter where they are executed. Accounting for the decentralized and dynamic approach fostered by liquid computing, with peers possibly joining and leaving at any time, the network fabric cannot rely on ahead-of-time knowledge for its establishment. Indeed, it shall only require the cooperation between the two involved clusters, which negotiate the configuration parameters necessary to set up (i) the secured communication channel and (ii) the proper mechanisms to guarantee any-to-any communication across the entire virtual cluster. Being the interconnecting clusters potentially under the control of different administrative domains, it is likely conflicts may arise, e.g., in terms of overlapping IP addresses or underlying networking solutions. The network fabric is expected to transparently handle all these issues, while virtually extending the local cluster network to the entire resource continuum, presenting a unique border-less addressing space. Supposing a central cluster *C* peered with *N* others, we foresee two main network topologies for data plane communications. First, a hub and spoke topology, with n direct interconnections between *C* and all the leaves. Conceptually simple, this solution requires all traffic between applications residing on peripheral clusters to flow through the central hub, potentially resulting in communication inefficiencies. Still, it may be appropriate when applications do not span across multiple remote clusters (e.g., the same application is replicated in multiple edge clusters), in case either the communication pattern or the underlying network match the star topology, as well as when traffic policies should be enforced from a single point of control. Second, an opportunistic mesh topology, providing full connectivity between all clusters hosting applications potentially communicating between one another, to avoid traffic tromboning. It is worth noting that peripheral clusters may in turn play the role of central clusters for different peering sessions, hence leading to completely dynamic and independent topologies, and potentially overlapped virtual clusters.

2.3.5 Storage and Data Continuum

When an application is spread across multiple clusters, stateful workloads require the access to persistent storage locations, which implies a data continuum across the virtual cluster. To this end, we foster the data gravity approach borrowed from well-established practice in the Big Data world [22]. According to it, data attracts the associated workloads (i.e., introducing additional placement constraints) rather than vice versa, to ensure the best performance in terms of reduced network traffic and latency, as well as to enforce storage locality, which represents a strong requirement to comply with law regulations. This paradigm allows also for the extension of traditional in-cluster stateful workloads replication mechanisms (e.g., databases) across the entire resource continuum, transparently achieving increased disaster recovery support. Information replication and synchronization might be further supported if more dynamism is desired, although requiring the exchange of data between potentially distant storage pools; hence, this is mostly suitable only in case of limited amounts of data.

Funded by Horizon Europe Framework Programme of the European Union

3 USE CASES AND SCENARIOS

This section describes a set of technical user stories that can take advantage of the computing continuum, which may go beyond the three use cases included in the FLUIDOS project (Table 1). They have been evaluated for suitability for each use case by the use case responsible within FLUIDOS. These stories are essentially technical-needs descriptions of the use cases and need to be supported by the technical work package outputs. When provided, use cases will then implement and deploy them for experimentation and evaluation.

Throughout the descriptions a generic actor is used, named "enterprise". This represents the different stakeholders of each use case and is the archetype actor.

The current set of use cases are as follows:

Table 1. Official use cases, with pilots, in FLUIDOS.

Name	Owner
Smart Viticulture	Terraview
Energy	RSE
Robotics	Robotnik

They all have the following needs from FLUIDOS. These needs are also aligned to the new technical developments of FLUIDOS.

3.1 CONTROLLING A LARGE NUMBER OF (EDGE) SERVERS/CLUSTERS

An enterprise owns a large set of edge servers, either completely independent or partially co-located together, which would like to control applications' life cycles in an easy manner. Administrators want to replicate the same service across all/a subset of the locations. Internet connection reliability might be an issue? Likely not in case of structured solutions (e.g., those managed by a telco), more probable with wireless connections (yet, these are probably associated with smaller/independent devices, which might not run Kubernetes in the first place).

3.1.1 Central Control Plane

One of the most common topologies is the **Central Control Plane**, where a central cluster, referred to as the management cluster, is responsible for multiple workload clusters. Although the management cluster is not critical, as existing pods and services continue to run even if

Funded by Horizon Europe Framework Programme of the European Union

the management cluster fails, updating applications during that time requires connecting to individual clusters. Therefore, it is recommended to have a highly available management cluster and a disaster recovery strategy.

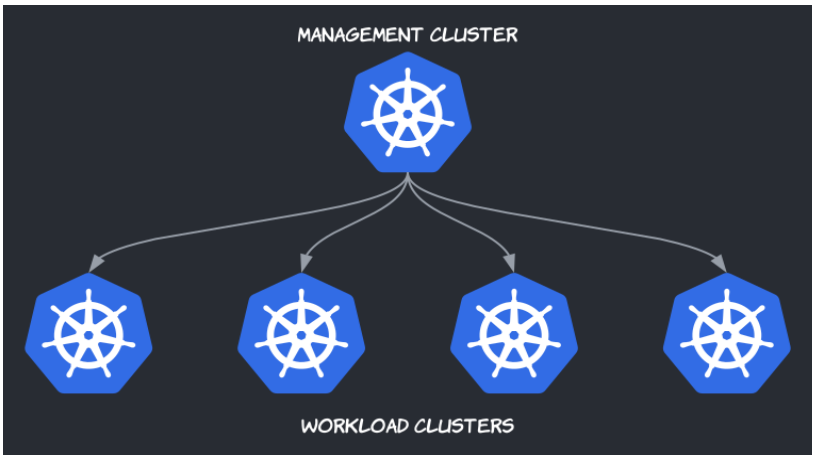


Figure 5. Possible deployment scenario: central control plane.

3.2 Connect applications/services hosted on different clusters

An enterprise owns multiple clusters, each hosting a given set of applications, e.g., for separation of concerns, due to historical/organizational reasons, to benefit from the managed services offered by the cloud provider. Yet, some applications might need to interact with others hosted on different clusters to provide their services, or access data (i.e., database) residing on other infrastructures. In a more general case, clusters might be controlled by different administrators/entities.

3.2.1 Cloud bursting

Cloud Bursting is a typical case in which a primary on-prem cluster with fixed resources targets both itself and a secondary cloud-based cluster with elastic resources. This use case is particularly interesting as long as on-prem resources are concerned, as those clusters are typically fixed and difficult to extend, hence need to find newer resources elsewhere. Instead, cloud-based clusters rely on a potential huge amount of resources, hence can leverage autoscaling mechanisms to acquire new resources when needed, e.g., to sustain an unexpected burst of workloads.

Funded by Horizon Europe Framework Programme of the European Union

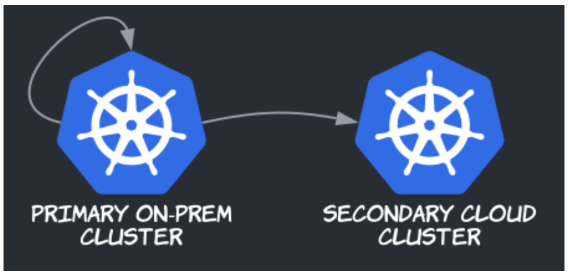


Figure 6. Possible deployment scenario: cloud bursting.

3.2.2 Decentralized Federation

Decentralized Federation enables organizations in different administrative domains to connect their control planes, facilitating the sharing of resources or delegation of operations without relying on a central cluster owned by a single organization. This topology is particularly useful for academic research platforms or vendor-managed deployments.

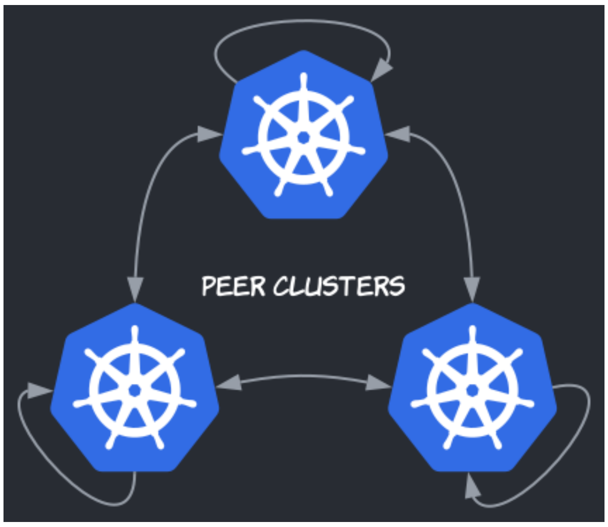


Figure 7. Possible deployment scenario: decentralized federation.

3.2.3 Multi-cluster high-availability and disaster recovery

An enterprise needs to deliver mission/safety critical services, which need cross-cluster high-availability and disaster recovery solutions to be in place. Thus, allowing to seamlessly migrate an entire application from an infrastructure to another in case of major outages. The most critical aspect concerns data synchronization.

3.2.4 Migration of applications across clusters

An enterprise has deployed a set of applications on a given cluster, and, for a set of reasons (e.g., agreement with a different cloud provider, cluster upgrade, …) needs to migrate

Funded by Horizon Europe Framework Programme of the European Union

temporarily or permanently all of them to a different cluster. The migration or relocation may be due to the triggering of intents associated with the application deployment.

3.2.5 Access to Data spaces

Different enterprises establish a relationship to share access to a given set of data, although each one maintains full control of its own part.

3.2.6 Secure Workload Container Independence

An enterprise wants to execute their workloads upon edge resources. The enterprise has different technologies to execute those workloads from VMs, containers, functions (FaaS), even plain OS processes. The application description needs to allow for application owners (the enterprise) to describe any workload container. Further the execution of this workload container needs to be secure and leverage the use of TEEs, this is especially the case on edge/on-premises devices that have lesser physical security and easier access by the public.

3.2.7 Intent-based Application Lifecycle Management

An enterprise needs to have a means to describe their application deployment clearly and concisely. They need to be able to deterministically specify where part of the application (a workload) needs to execute regarding location within the compute continuum. These specifications should also be loosened to be flexible. This flexibility should be given by intents in the case that location requirements are driven more from the functional aspects (e.g., latency). As such the system can better reason with the placement of these workloads. Further, other intents (aims) can be composed and resolved by the system. Such intents that are of interest and need by the use cases are Cost-based intents e.g., deploy this part of the system on the cheapest resources, move this workload when it becomes too expensive at the provider to another suitable provider. Further, energy-based intents are needed for example always aim to have application workloads executing on energy efficient hardware.

Funded by Horizon Europe Framework Programme of the European Union

4 REQUIREMENTS

This section lists the preliminary requirements that have been collected (1) from the FLUIDOS use case owners (TER, ROB, RSE) and (2) through external consultation, with respect to the objectives that are perceived as unique values by the FLUIDOS stakeholders.

While this activity is still a work-in-progress at the time of writing this deliverable, in the future those requirements will be translated into technical objectives, which will be used to assess the success of this project.

I want…	So That…
That FLUIDOS is aware of the device (e.g., robot) battery level, and power consumption	I can remove or do not add computational workload in a almost empty battery robot, or heavy lifting robotic operation
That FLUIDOS is aware of wireless network quality	I can plan if can switch safely a load from cluster to another grading the network quality during the mission
That FLUIDOS is aware of the state of robot (in mission/idle/moving platform/moving arm/charging)	FLUIDOS can automatically and manually plan which the workload across the robot clusters taking in account robot state, and each state have different needs and goals
That FLUIDOS allow me to switch k8s workload from cluster to another by request	I can ask to switch off k8s workload from k8s (cluster) to specific one (or FLUIDOS choose) by request of the fleet manager
I want to build a robot map outside the robot. This process is takes long and can be done offline	I can make this CPU and/or GPU extensive process on the cluster that maximizes the time and working time of the robots
Easy installation and low maintenance of the FLUIDOS system on the edge, cloud or robots	I can deploy fast anything I have over the edge and without deeper knowledge
An easy way to define the workloads taking in account all the points above	I can deploy, update, rollback, change the orchestration rules so I can sell FLUIDOS to non-technical customer (web based and yaml, etc..)

Funded by Horizon Europe Framework Programme of the European Union

5 STATE OF THE ART

This Section reports the state-of-the-art of algorithms, technologies, and software, updated to the writing time of this Deliverable, separating the different problems that are addressed by FLUIDOS, such as the creation of the computing continuum, its security, and more.

5.1 COMPUTING CONTINUUM

The idea of providing unified computing resources within a single continuum has been around for a few years, with multiple proposals both from the scientific community and enterprise-driven open-source projects. The common idea is to provide a unified view of multiple distributed clusters either belonging to a single entity or located in a specific geographic area. Scientific papers tackle the problem from different points of view, proposing full-fledged solutions or Proof-of-Concepts (PoCs), either by focusing more on a vanilla approach with respect to the de-facto standard container orchestrator Kubernetes or by implementing their specific set of APIs.

This Section presents the three most promising open-source projects that are most suited as a starting point for the liquid computing idea. More experimental projects that, although not the best choice to base FLUIDOS upon, can nevertheless provide nice ideas that can be possibly integrated and further developed in FLUIDOS, and additional related technologies is provided in the Appendix (Section 11).

A final comparison will be provided, to allow the reader to better compare the respective advantages and disadvantages.

5.1.1 KubeEdge

KubeEdge [23] is an open-source tool that enables the orchestration of containerized applications on Edge devices using Kubernetes. It provides infrastructure support for network, application deployment, and metadata synchronization between the Cloud and Edge. Essentially, KubeEdge allows developers to write HTTP or MQTT-based applications, containerize them, and run them either at the Edge or in the Cloud, depending on their use cases.

Funded by Horizon Europe Framework Programme of the European Union

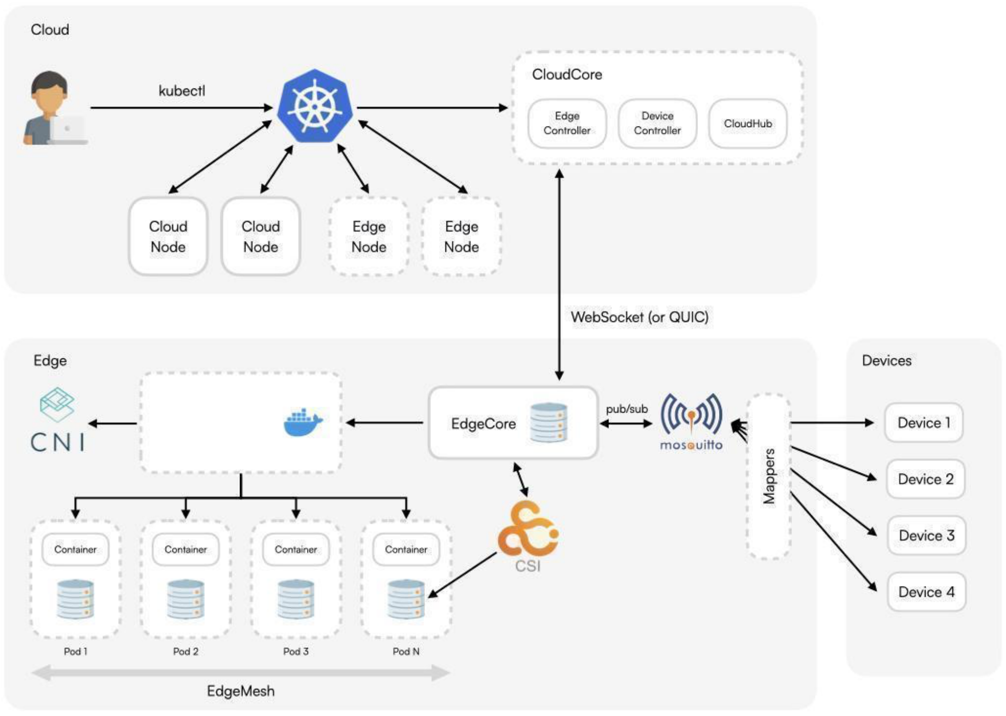


Figure 8. KubeEdge architecture.

The Cloud side of KubeEdge is considered the master cluster, whether on-premises or managed by cloud providers like Amazon or Google. This side connects with edge devices remotely, deploying pods efficiently on them. Core components on both Cloud and Edge sides are necessary to facilitate communication, with the most crucial ones being **CloudCore** and **EdgeCore**:

- **CloudCore**: This component runs in the cloud and is responsible for managing the overall KubeEdge system. It is the gluing layer between Kubernetes API server and the KubeEdge edge part. Moreover, it communicates with nodes and other components at the edge to orchestrate the deployment and management of containerized applications and devices on the edge.
- **EdgeCore**: This component runs on the edge device and is responsible for managing the deployment and execution of containerized applications on the edge device. It communicates with the KubeEdge Controller to receive instructions and to report the status of the applications running on the edge device. It is also responsible for detecting and registering the devices and sensors connected to the edge device with the KubeEdge system. It enables applications running on the edge device to access the data and functionality of these devices and sensors.

To enable adding edge devices as nodes of the Kubernetes cluster, the CloudCore component's installation is required on the cloud side. This component consists of three

Funded by Horizon Europe Framework Programme of the European Union

modules: CloudHub, EdgeController, and DeviceController. CloudHub acts as a mediator between EdgeController, DeviceController, and Edge side for communication. EdgeController bridges the Kubernetes API server to EdgeCore, while DeviceController manages device metadata/status in a dynamic manner.

On the edge side, the installation of a container runtime such as Docker or containerd and EdgeCore components is necessary for communication with the cloud. EdgeCore has six modules, including EdgeD and EdgeHub, with a reduced memory footprint, responsible for managing the lifecycle of pods and serving as a communication link between the edge and cloud, respectively.

In summary, KubeEdge facilitates the orchestration of edge devices by installing components on both the cloud and edge sides. While KubeEdge has reduced resource requirements on edge nodes, it does not provide network, storage, and service fabric across the entire virtual infrastructure. Consequently, it is more suitable for IoT-oriented applications that run at the edge, gather local data, and send it to other services running on the cloud side for further processing and storage.

5.1.2 Karmada

Karmada (Kubernetes Armada) [24] is a management system for Kubernetes that enables cloud-native applications to run on multiple clusters and clouds without the need to modify the applications. Karmada utilizes Kubernetes-native APIs and advanced scheduling capabilities to provide a seamless, multi-cloud Kubernetes experience. Its aim is to automate multi-cluster application management in hybrid cloud scenarios with centralized multi-cloud management, high availability, failure recovery, and traffic scheduling.

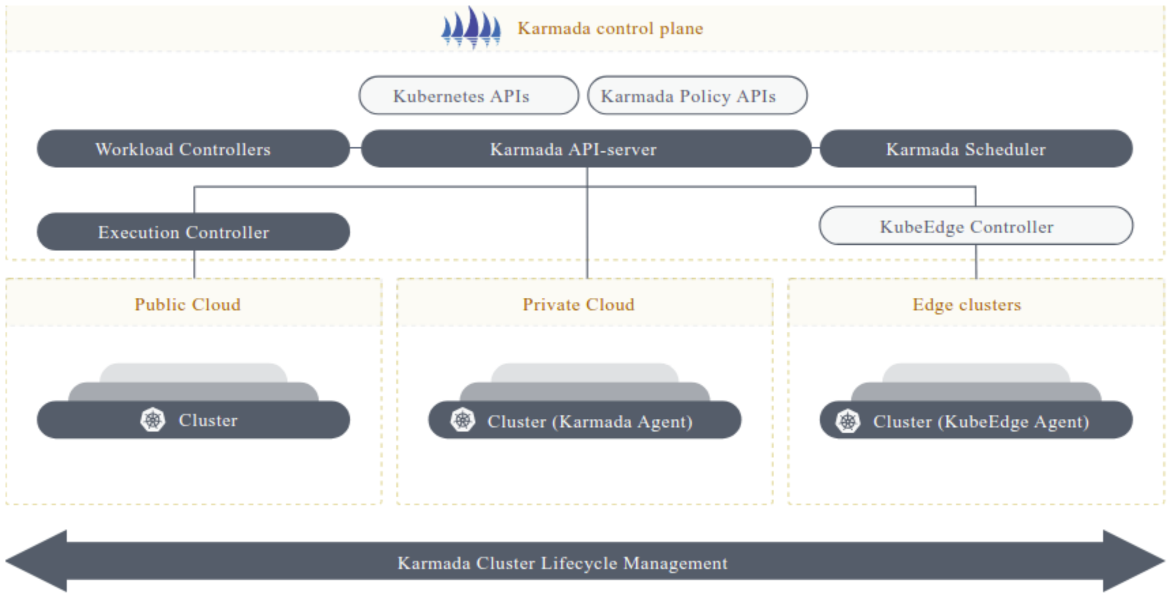


Figure 9. Karmada overview.

Funded by Horizon Europe Framework Programme of the European Union

Karmada is compatible with the Kubernetes native API and can seamlessly upgrade from a single-cluster to multi-cluster while integrating with the existing K8s toolchain. However, most of its capabilities require the use of new CRDs, which requires users and dev-ops to learn a new way to interact with their clusters, as traditional Kubernetes primitives cannot be used to control the virtual continuum.

Karmada has three built-in policies for deployment scenarios and multiple scheduling policies for cluster affinity, multi-cluster splitting/rebalancing, and multi-dimension HA. It also provides centralized management for clusters in public cloud, on-premises or at the edge.

Karmada offers various concepts and resources that manage the multi-cloud environment, including a Resource Template, Propagation Policy API, and Override Policy API. Its overall architecture includes the Karmada Control Plane, which includes the Karmada API Server, Karmada Controller Manager, Karmada Scheduler, and an etcd database.

Karmada supports safe isolation, multi-mode connection, and multi-cloud. It also offers support for cluster distribution capabilities under various scheduling strategies and allows for differential configuration through OverridePolicy. Karmada further offers rescheduling functionalities with components like Descheduler and Scheduler-estimator that trigger rescheduling based on instance state changes in member clusters.

Karmada supports cluster failover, cluster taint settings, and global uniform resource view. It also offers unified authentication with a single API access entry and access control that is consistent with member clusters.

Karmada is a tool that facilitates the administration of multiple Kubernetes clusters using the master-worker model. However, this model presents a challenge as it does not allow for a federation of clusters at equal levels. Instead, one cluster must always take the lead in coordinating the others. Additionally, Karmada does not include a default network fabric. To enable communication between pods in the clusters, Karmada relies on Submariner. This means that Submariner must be installed in each cluster to ensure seamless communication.

5.1.3 Liqo

Liqo [25] is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premises, cloud and edge infrastructures. While its focus is on cloud-oriented datacenter and servers, it could be used also on individual devices (e.g., coupled with lighter Kubernetes distributions, such as K3s), although it may not be appropriate to support features such as real-time scheduling and such.

Among its distinctive features, Liqo encompasses four primary capabilities, namely peering, offloading, network fabric, and storage fabric.

- **Peering**: it involves one-way relationships between two Kubernetes clusters, which can act as providers and consumers in multiple peerings, and involves four tasks: authentication, parameters negotiation, virtual node setup, and network fabric setup.

Funded by Horizon Europe Framework Programme of the European Union

The network fabric is configured to establish a secure cross-cluster VPN tunnel, enabling seamless communication between local and remote cluster pods.

- Offloading: the virtual node abstraction enables workload offloading by spawning a virtual node that represents and aggregates resources from a remote cluster, which is fully compliant with standard Kubernetes APIs and supported by Liqo, that creates twin namespaces for offloaded pods, supports both stateless and stateful pods, and ensures remote pod resiliency through a custom resource, while propagating and synchronizing selected control plane information into remote clusters for seamless execution of offloaded pods.
- Network fabric: it enables pods to communicate with each other across multiple clusters, even if they have overlapping network spaces.
- Storage fabric: it simplifies multi-cluster scenarios by postponing storage binding until the first consumer is scheduled and using data gravity to ensure pods requesting existing storage pools are scheduled onto the cluster hosting the corresponding data.

Liqo is a software tool that enables the management of multiple Kubernetes clusters. It can operate using either the master-worker paradigm or not. The first scenario allows Liqo to use one cluster to coordinate the others, while the second one creates a federation of clusters in which each one is at the same level as the others without any hierarchy. Liqo also comes with a built-in network fabric, enabling communication between pods when they are offloaded to another cluster.

5.1.4 Comparison summary

This section provides a compact comparison between the most promising solutions presented in this document. The comparison is presented in the form of a table, to summarize the most important features, giving an easier way for the reader to understand the similarities and differences between solutions.

Table 2. Most promising projects to build the computing continuum.

	KubeEdge	Karmada	Liqo	Cloudlets	FLEDGE	CRDTs DB
Use K8s	Partially	Yes	Yes	No	No	Yes
Cluster or Node	Node	Cluster and Node	Cluster	-	Node	Cluster
Networking	External	External	Included	-	Included	-
Resource consumption	Low	High	High	Low	Low	-
Status	Well-established	Well-established	Well-established	Experimental	Experimental	Research
# GitHub stars	5800	3200	820	-	-	-
# GitHub contributors	267	170	40	-	-	-

Funded by Horizon Europe Framework Programme of the European Union

1st GitHub commit	Sept 2018	Nov 2020	Nov 2019			

The above table shows that, among the most interesting projects, only three are well-established, actively supported and hence can be proficiently used as a foundation to build the FLUIDOS computing continuum, namely KubeEdge, Karmada, and Liqo. However, KubeEdge's focus is more for the "far" edge of the network, particularly tiny individual devices. Hence, from the point of view of cloud native technologies (at the edge / cloud), the most appropriate choices are Karmada and Liqo. Karmada provides a specialized (hence, not compatible with Vanilla Kubernetes) API server to process requests and enforce them onto clusters but does not include network fabric in its functionalities and cannot handle scenarios different from the master-worker one. On the other hand, Liqo is almost 100% compatible with vanilla Kubernetes, it does not mandate a master-worker scenario (although it can be supported) and can create a federation where each cluster is at the same level as others, allowing a different topology between clusters. Additionally, it provides a network fabric, making it possible to get multi-cluster and pod connectivity features from a single solution.

Based on this analysis, Liqo represents the most promising technology to base FLUIDOS upon thanks to its ability to provide a topology between clusters instead of a hierarchical scenario, with the possibility to be integrated with KubeEdge for "far" edge scenarios. In fact, KubeEdge is the ideal choice for "tiny" edge devices since it enables their integration into Kubernetes clusters, facilitating their management and deployment of pods on edge devices, which is useful for various purposes such as reducing distance from users, with minimal resource requirements.

Certainly, there are certain gaps that require addressing for enhancing the effectiveness of the current solutions presented. For a solution that strives to establish a computing continuum, it is crucial to ensure that all clusters are managed equally, while also allowing for the creation of hierarchies if necessary. A missing element relates to the exchange of details regarding the resources available in different clusters, which impedes negotiation and contractual agreements. Additionally, ensuring the security of all the aforementioned operations is a substantial gap that needs to be filled.

5.2 Security and Privacy

The evolution of computing paradigms is slowly breaking the boundaries of data centers, widening the infrastructure's scope to include edge and micro-edge nodes. The complexity of cloud-to-edge environments, their heterogeneous nature, as well as the intricacies of cloud-native applications and their management systems have raised concerns within the security and privacy community [38]. FLUIDOS proposes to go beyond the current evolution and envisions a future in which it would be possible to dynamically create cloud-to-edge computation clusters among peering entities. In such a scenario, the traditional concept of

Funded by Horizon Europe Framework Programme of the European Union

the security perimeter is expected to disappear, in favor of a zero-trust approach to security [39], in which nothing, even inside an organization's network, can be trusted.

In this respect, it is worth noting how, the idea of having perimeters blurs from the security point of view, but it is still valid when another point of view is considered. For instance, according to Section 1.1, the three characteristics of *Deployment*, *Communication* and *Resource Availability* transparency still hold within a given perimeter, while cannot be achieved at large, outside the current (virtual) cluster.

FLUIDOS plans to investigate zero-trust methods and algorithms to guarantee security and privacy in storing, transmitting, and processing data in the envisioned cloud-to-edge continuum. To this account, the consortium identified the following main areas of investigation: **identity management** (authentication, authorization and key management) to ensure that legitimate parties have the right access to the right resources, **isolation and trusted computing** to provide secure environments to run workloads in environments that are not owned, **anomaly detection** to protect both hosts and guests, and **security orchestration** to manage the security posture in an infrastructure that is changing over time. The following subsections provide a preliminary overview of the state-of-the-art solutions in the above-mentioned areas.

5.2.1 Identity Management

The FLUIDOS continuum could benefit from a distributed approach to identity management, in which the user perspective and its Self-Sovereign Identity (SSI) are possibly central, as explained in C. Allen [40] and envisioned by the European Union with several initiatives as the European Self Sovereign Identity framework (eSSIF) [41].

Indeed, current centralized access control systems, based on a trusted third party, have major limitations, including a high cost of trust and the risk of a single point of failure. Latest research proposes to employ decentralized identifiers (DIDs) [42] and verifiable credentials (VCs) [43] as a highly distributed and lightweight alternative to well-known and established authentication methods. The blockchain technology can be used both as a replacement for the registration authority (A. Mühle et al. [44]) and as an authorization scheme for distributed services (as in Yu L. et al. [45], which specifically targeted mobile cloud services). The scheme proposed by Yu et al. uses smart contracts to provide a dynamic update of access permissions and reduces the storage overhead by storing only one transaction in the blockchain to record users' access privileges on different service providers.

N. Fotiou et al. [46] provide a proof of concept of a DID registry and a distributed Policy Decision Point (PDP) and Policy Enforcement Point (PEP). In G. Peterson [47] PEPs are proposed to enable fine-grained, decentralized security policy decisions through languages such as Extensible Access Control Markup Language (XACML). The security policy manager bundles these decisions into standards-based XML documents that can be transported and

Funded by Horizon Europe Framework Programme of the European Union

consumed across many disparate parts in the system. This lets a PEP query policy decision points (PDPs) to make authorization decisions in a highly distributed way.

With respect to management of cryptographic keys in the cloud-to-edge scenario, S. Kahvazadeh et. al. [48] propose a Distributed Key Management and Authentication (DKMA) solution that tries to overcome the issues that a centralized key distribution method fails to address. As a centralized alternative, Key Management Systems (KMSs) and associated cryptographic services could also be provided as cloud-based services, such as the case of AWS Key Management Service (KMS) [49], Microsoft Azure Key Vault [50] and Google Cloud KMS [51]. These solutions offer scalable and cost-effective key management services that can be easily integrated with cloud environments. Containerization is also becoming a popular deployment model for KMS solutions, allowing for greater flexibility and portability across different environments.

5.2.1.1 Towards the FLUIDOS vision

For identity management, the project will analyze the available solutions in literature and commercial products to choose the ones that best suit FLUIDOS's characteristics. To identify the nodes, we prefer the use of identifiers that can be managed in a decentralized manner, such as the Decentralized Identifiers recommended by W3C, rather than centralized identifiers and credentials. Additionally, we will use verifiable credentials that can be generated in different FLUIDOS domains while maintaining their independence. This approach allows us to maintain a decentralized architecture that allows for multi-ownership, ensuring that each actor has control over their own resources.

To store the identity information and contracts between the peers, we will use distributed storages such as Distributed Ledgers. We will avoid using KMSs and cryptographic services in the cloud that would generate unwanted dependencies. By using these elements, we can explore the possibilities that their different configurations can provide in a fluid topology of FLUIDOS to face problems like preserving the privacy of the contracts between peers.

5.2.2 Isolation and Trusted Computer

Trusted Execution Environments (TEE) and Secure Enclaves are technologies that provide a secure environment for executing sensitive and critical applications, such as mobile banking, digital signatures, and secure communications. The Trusted Execution Environment (TEE) technology is a rapidly evolving field, with new developments and innovations happening all the time. Technological advancements in the field of TEE range from Hardware-based TEEs (e.g., AMD Secure Encrypted Virtualization [52], Intel TDX [53]) to Cloud-based TEEs (e.g., Amazon Web Services (AWS) Nitro Enclaves [54], Google Cloud Confidential Computing [55]) and Virtualized TEEs (e.g., Samsung KNOX Virtual Private Space (VPS)).

The research community is rather active in the field: for instance, Gu et al. [56], propose an SDK to enable enclave migration on the cloud, while Alder et al. [57] build on a secure live

Funded by Horizon Europe Framework Programme of the European Union

migration of enclaves on untrusted clouds and use a per-host dedicated enclave to migrate the persistent state of an enclave.

Software-based developments of the concept of TEEs include the so-called "Confidential Container" [58], that is a secure, isolated environment for storing and processing sensitive information. Confidential Containers are similar to Trusted Execution Environments (TEEs), as they provide a secure environment for sensitive information; however, they focus specifically on the storage and management of confidential data.

Of particular interest for the FLUIDOS project is the TEE orchestration, which refers to the management and coordination of multiple Trusted Execution Environments (TEEs) to provide a secure and efficient environment for executing sensitive and critical applications.

TEE orchestration involves various tasks, such as the provisioning and deployment of TEEs, their management and monitoring, the establishment of secure communications among TEEs, load balancing and resource allocation, compliance, and security management. TEE orchestration is available today with platforms like Fortanix [59] and Red Hat OpenShift [60]. Moreover, the majority is designed to support a specific type of TEE.

5.2.2.1 Towards the FLUIDOS vision

FLUIDOS will need to extend this orchestration capabilities along the continuum to enable a seamless management of sensitive workloads in the cloud, on-premises, or on the edge of the network. It should also provide a unified management interface for deploying and managing TEEs across different secure computing environments (AMD-DEV, Intel SGX, Arm Trustzone…). Consequently, the TEEs orchestration within FLUIDOS is expected to combine multiple secure runtimes.

5.2.3 Proactive and reactive container defense

In a zero-trust environment, each workload should be secured from threat actors as well as honest but curious infrastructure providers. At the same time, protection should be granted to compute nodes exposed in FLUIDOS and to the applications they are running. This can be done either proactively (before deployment) by securing single containers, or reactively (at runtime) by detecting anomalies in the containers' behavior.

5.2.3.1 Proactive container defense

With respect to the former, within FLUIDOS we will rely on containers not only to develop user applications but also to run the various components that make up a FLUIDOS node.

Containers provide a lightweight form of virtualization at the cost of a weaker isolation with respect to virtual machines (VMs). As shown in literature, adversaries can attack container ecosystems to discover information leakage channels that can be exploited from within containers [61][62], gain elevated privileges in the host [63], slow down the containers'

Funded by Horizon Europe Framework Programme of the European Union

execution or disrupt their availability [64], and impersonate containers [65] (among others). There are several fundamental reasons why these attacks are possible. Firstly, the shared kernel-resource model used by containers offers less isolation than VMs and significantly increases their attack surface. This can increase the impact of attacks because a single vulnerability in the kernel can allow adversaries to compromise co-resident containers, or even worse, the host. Secondly, there are public repositories of container images that are widely used today - and are even used as a basis for creating other images - which, based on the results of various research works, contain a high number of vulnerabilities. Thirdly, the attack surface of the container ecosystem remains largely unexplored – especially when it comes to privacy attacks. Four, some security mechanisms to protect containers – based on standard security features and network access control policies of Linux – have been shown to be vulnerable to attacks [66][67]. Last but not least, the majority of security mechanisms require network operators to manually configure them. The latter is typically a laborious task that is prone to misconfigurations, especially in such dynamic environments.

5.2.3.2 Towards the FLUIDOS vision

From the previous discussion, it is clear how impossible it is to completely protect the container ecosystem against security attacks. In these scenarios it is therefore important to have containment mechanisms in place to minimize the consequences of attacks as much as possible when they occur. In this line of work, we are currently designing a tool capable of automatically identifying the minimum set of privileges that containers need to function, with the aim of significantly reducing the attack surface of the kernel. According to the NIST security guidelines [102], kernel attack surface reduction techniques are a promising approach to strengthen security in container environments. In parallel to the above, from a privacy perspective, we are exploring various types of side-channels that may arise in a confidential computing scenario to leak sensitive information about containerized applications. More specifically, we are studying the feasibility of attacks whereby adversaries try to obtain sensitive information about containerized applications running inside a TEE solely from the system calls they invoke. With this work, we aim to demonstrate that it is crucial to protect the interfaces of containerized applications running inside the TEE with the "outside world."

5.2.3.3 Reactive container defence

With respect to reactive defense, even in the case in which the container runs in a TEE, the host must provide containers with access to system calls, an interface that might be exploited to perform attacks. To this account, El Khairi et al. [68] propose a novel anomaly-based Intrusion Detection System (IDS) that relies on monitoring heterogeneous properties of system calls. Attacks might target other components and processes of the FLUIDOS continuum. Cloud and edge nodes shall be equipped with advanced IDS systems able to countermeasure different kinds of attacks, such as LUCID [69], a lightweight neural network proposed to detect Distributed Denial of Service (DDoS) attacks, or the platform for automated detection and mitigation of DDoS attacks proposed by Demoulin et al. [70]. Lawal

Funded by Horizon Europe Framework Programme of the European Union

et al. [71] proposed an anomaly mitigation framework for IoT that resorts to fog computing to guarantee faster and accurate detection.

5.2.3.4 Towards the FLUIDOS vision

Even though the body of literature targeting threat and intrusion detection and mitigation is huge, FLUIDOS introduces challenges that are seldom explored in these works. For instance, the project assumes that resources at the edge will be shared with different potential acquirers based on opportunistically created agreements. Depending on the type of service provided, it could be assumed that the operating system on the node is not trustable, or that the workload is not, and therefore, it might be the case that the project needs to ensure that network traffic and/or system calls are not being used to attack the host machine. However, given the limited amount of resources available at the edge, it is of the utmost importance to limit the computational footprint of the monitoring and detection processes while guaranteeing an adequate level of accuracy. Another venue of investigation might concern the use of images by different tenants insisting on the same resources.

5.2.4 Intent-based Security Orchestration

The works mentioned above provide detection and mitigation methods for specific attacks and under given conditions. However, managing the security posture in the cloud-to-edge continuum proposed by FLUIDOS requires effective and automated coordination of multiple security services over a dynamically changing substrate infrastructure. Such an ambitious objective can be achieved through automated, intent/policy-based security orchestration. To this account, Zarca et al. [72] propose an architecture to orchestrate security policies in proactive and reactive ways, so the system is able to react dynamically with different countermeasures (filtering, forwarding, IoT honeynet, IoT management or service management among others) to different kinds of attacks (e.g., DoS, SlowDoS, DDoS, data tampering or other abnormal behaviors), according to the current status of the infrastructure. J. Kim et al. [73] defined a security framework in the field of cloud computing to provide an advanced system that interprets and translates intents to deploy security policies without the need of any security intervention from the operator. However, the framework requires human intervention, does not deal with the dynamic federation of infrastructures, and does not provide automated detection and mitigation of attacks. D. Bringhenti et al. [74][75] proposed a new methodology for automated orchestration and configuration of distributed firewalls in a virtualized network. The proposed methodology is based on a formal model that assures that the final solution satisfies the security requirements (correctness by construction). Valenza et al. [76] define two abstract languages to represent Network Security Intents at a high-level or medium-level of abstraction.

Among the security techniques that require efficient management in the continuum, cyber deception represent a notable candidate for FLUIDOS, as it provides an excellent complement to anomaly detection to reduce false positives as well as a priceless source of cyber threat intelligence information, but it should be carefully orchestrated to avoid needless

Funded by Horizon Europe Framework Programme of the European Union

resource consumption. For many years, the go-to mathematical framework for solving this problem has been game theory, where it is modeled as a game between the attacker and defender party, each characterized by its own reward function. Ferguson-Walter et al. [77] point out that one-shot games do not manage to properly capture the possible interactions between the two parties and therefore propose to use multi-stage games for this purpose. Other studies, instead, propose methodologies to address decoy placement leveraging machine learning models, starting from the assumption that it is possible to learn from the environment itself the best placement. Amin et al. [78] propose a dynamic honeypot placement method based on a Hidden Markov Model, where decoys are placed alongside the most probable attack paths according to previous attack records. However, none of these works consider the limited availability of resources that is peculiar of edge environments. In addition to that, most of these works and some of the solutions commercially available (e.g., Metallic ThreatWise [79]) rely upon pre-built applications that are usually deployed as standalone elements within the defender environment. However, as Osman Amr et al. [80] propose, a deception mechanism could be based on clones of existing containers in a cloud-native environment. This is practically an unsolved challenge, as it requires a complex procedure to clone, isolate and deploy deceptive containers.

5.2.4.1 Towards the FLUIDOS vision

The works mentioned above have taken approaches that align with FLUIDOS's intent-driven characteristic. However, these approaches were tested in closed infrastructures, specific scenarios, and with limited sets of cases using algorithms. To tailor and extend these solutions for FLUIDOS, we need to enhance the security orchestration through the node security orchestrator, which will collaborate with the service orchestrator and other FLUIDOS components to elaborate the user intent and ensure the required security level.

We will design novel security orchestration algorithms that take into due account the extreme dynamicity of the infrastructure that is envisioned in FLUIDOS. In order to complement the proposed anomaly detection and mitigation strategies, we will propose a novel orchestrated, cloud-native and resource-aware cyber deception solution.

With this foundation, we will test a set of security orchestration algorithms and techniques, analyze their behavior and security levels in a fluid computing continuum. This will allow us to evaluate and improve the security of the FLUIDOS *liquid computing* continuum.

5.3 ENERGY-AWARE COMPUTING CONTINUUM

This section presents the state of the art with respect to the energy-aware computing continuum. Specifically, we first detail the main available technologies to collect energy-related metrics of the devices participating in the continuum resource-sharing process. We

Funded by Horizon Europe Framework Programme of the European Union

then provide an overview of the most promising approaches of major cloud providers in reducing the overall carbon and energy footprint for computing devices.

5.3.1 Energy measurement tools

The following sections will analyze the available tools to collect power consumption metrics from Linux-based systems.

5.3.1.1 Intel RAPL

RAPL (Running Average Power Limit) is a reporting interface for cumulative energy usage of multiple system-on-chip (SoC) power domains. The RAPL energy reporting capability has been present on Intel SoC devices for several generations, and energy reporting is industry standard practice. This energy information is used by Intel processors for internal SoC management functions, such as controlling SoC power limits in conjunction with Intel Turbo Boost Technology[2].

At runtime, certain privileged software may execute platform power and temperature management. Energy information from RAPL may be used by such applications to adjust system performance or track power use. Energy data is occasionally utilized in server systems to perform rack-level power management and efficiency loading across units.

Platforms in RAPL are separated into domains for fine-grained reporting and control. A RAPL domain is a physically significant power management domain. The exact RAPL domains offered in a platform differ depending on the product category. RAPL implements four power domains (Figure 10):

1) Package Domain: This power domain represents the power consumption of the CPU's complete package, including cores and additional components (i.e., integrated graphic card).
2) DRAM Domain [101]: This power domain accounts for the power consumption of the DRAM. It is only accessible on servers.
3) PP0/Core Domain (PowerPlane 0): is used to monitor the power of the CPU cores only.
4) PP1/Graphic Domain (PowerPlane 1): is used to monitor the power of the CPU's graphic component alone. Because of this, it is only available for non-server architectures. Graphic components are not included in Intel server designs.

NOTE: The reported power consumption at the Package domain is: PPKG=PP0+ PP1.

[2] Although the RAPL interface has been designed by INTEL, some AMD chips are also compatible. AMD Fam17h supports an MSR interface that is semi-compatible with RAPL. Older AMD supports the Application Power Management (APM) interface. The CPU compatibility table is available at https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html.

Funded by Horizon Europe Framework Programme of the European Union

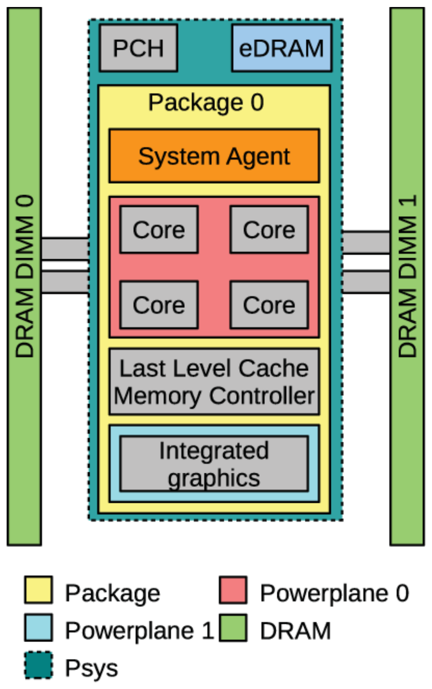


Figure 10. Different power domains in RAPL.

Each RAPL domain supports:

- ENERGY_STATUS for power monitoring.
- POWER_LIMIT and TIME_WINDOW for controlling power.
- PERF_STATUS for monitoring the performance impact of the power limit.
- RAPL_INFO for information on measurement units, the minimum and maximum power supported by the domain.

In addition, RAPL has 32-bit performance counters for each power domain to monitor energy use and overall throttled time.

5.3.1.2 ACPI interface

The Advanced Configuration and Power Interface (ACPI) specification is an open standard that was created by a group of hardware and software companies including HP, Intel, Microsoft, Phoenix e Toshiba. It defines standard interfaces that allow operating systems to configure motherboard devices and control power. ACPI may be used for specific hardware components as well as the full system. Furthermore, it monitors the state of the system and utilizes power management techniques by altering the CPU running frequency and putting unnecessary components to sleep.

ACPI is intended to allow the operating system to control each hardware component. Prior to the development of ACPI, power management was handled via Plug and Play (PnP) and Advanced Power Management (APM) subsystems, which were implemented in hardware and

Funded by Horizon Europe Framework Programme of the European Union

hence were less versatile in terms of management capabilities. ACPI, on the other hand, is implemented in the OS layer and thus gives greater freedom for component management, as well as being hardware independent (if the hardware meets the ACPI standard).

The ACPI allows the operating system to push certain hardware devices to a low power consumption state when they are not in use. Similarly, if the operating system determines that the applications do not demand a huge number of resources, ACPI can move the entire environment to a low power consumption mode.

The ACPI specification details different device states to represent the execution of the system. Specifically, they can be classified into:

- **G states**, representing the system's global execution state (e.g., working, sleeping, …).
- **D states**, representing the device-dependent states (e.g., fully ON, …).
- **C states**, representing the CPU power states. They detail the different CPU execution states (e.g., operating state, halt, sleep, …).
- **P states**, representing the operating frequency of the CPU cores (e.g., maximum frequency, frequency scaled, …).

Using ACPI it is thus possible to interact with the different states of the device and collect relevant metrics on the device execution. Furthermore, it is possible to modify the various states according to energy-aware management algorithms.

5.3.1.3 IPMI interface

IPMI is a collection of computer interface requirements for a self-contained computer subsystem that offers administration and monitoring capabilities independently of the host system's CPU, firmware (BIOS or UEFI), and operating system. IPMI is a set of interfaces used by system administrators for out-of-band administration and monitoring of computer systems. For example, IPMI allows the administration of a machine that is switched off or otherwise unresponsive to connect to the hardware over a network rather than an operating system or login shell.

The use of a standardized interface and protocol allows IPMI-based systems management software to control many different servers. IPMI, as a message-based, hardware-level interface specification, functions independently of the operating system (OS) to enable administrators to remotely control a system in the absence of an operating system or system management software. As a result, IPMI functionalities can be used in one of three scenarios:

- Before an OS has booted (allowing, for example, the remote monitoring or changing of BIOS settings).
- When the system is powered down.
- After OS or system failure – the key characteristic of IPMI compared with in-band system management is that it enables remote login to the operating system using SSH.

Funded by Horizon Europe Framework Programme of the European Union

IPMI messaging may be used by system administrators to monitor platform status (such as system temperatures, voltages, fans, power supplies, and chassis intrusion).

The intelligence in the IPMI architecture is provided by the baseboard management controller (BMC). It is a customized microcontroller that is integrated on the motherboard of a computer, most commonly a server. The BMC is in charge of overseeing the interaction between system management software and platform hardware. BMC has its own firmware and RAM.

Temperature, cooling fan speeds, power status, operating system (OS) status, and other characteristics are reported to the BMC by various sensors embedded into the computer system. The BMC monitors the sensors and can transmit network warnings to a system administrator if any of the metrics exceed pre-set thresholds, signaling a potential system failure.

5.3.1.4 KEPLER

Kepler (Kubernetes-based Efficient Power Level Exporter) [81] is a Kubernetes-based solution for energy and power consumption monitoring on a cluster of servers. It uses eBPF to probe CPU performance counters and Linux kernel tracepoints and collect the exact CPU consumption of the different processes running on the system. Kepler then correlates such information with the power consumption metrics of the device to compute the contribution of the different processes to the overall power consumption of the device.

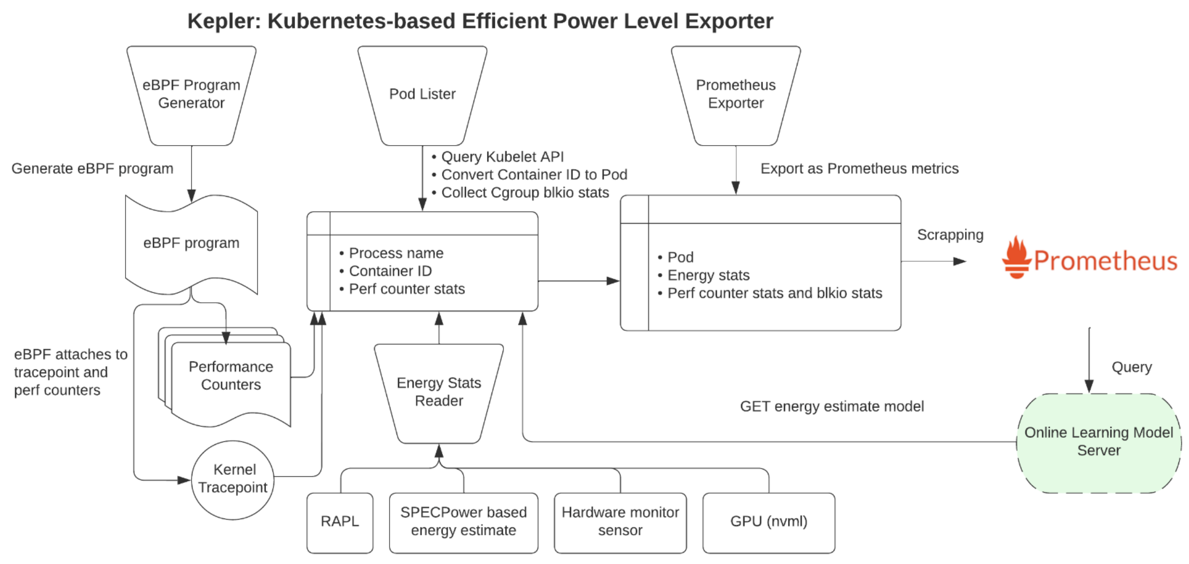

Kepler: Kubernetes-based Efficient Power Level Exporter
eBPF Program Generator
Pod Lister
Prometheus Exporter
Generate eBPF program
• Query Kubelet API
• Convert Container ID to Pod
• Collect Cgroup blkio stats
Export as Prometheus metrics
eBPF program
• Process name
• Container ID
• Perf counter stats
• Pod
• Energy stats
• Perf counter stats and blkio stats
Scrapping
Prometheus
eBPF attaches to tracepoint and perf counters
Performance Counters
Energy Stats Reader
Query
GET energy estimate model
Online Learning Model Server
Kernel Tracepoint
RAPL
SPECPower based energy estimate
Hardware monitor sensor
GPU (nvml)

Figure 11. Kepler: overview of the main workflow.

Specifically, power consumption metrics are collected using the tools detailed in the previous section (INTEL RAPL, ACPI, IPMI) if available. If not available or incomplete, Kepler relies on a pre-trained neural network to generate an estimate of the power consumption, the Kepler Model Server. The main feature of the Kepler Model Server is to return a power estimation model corresponding to the request containing target granularity (node in total, node per

Funded by Horizon Europe Framework Programme of the European Union

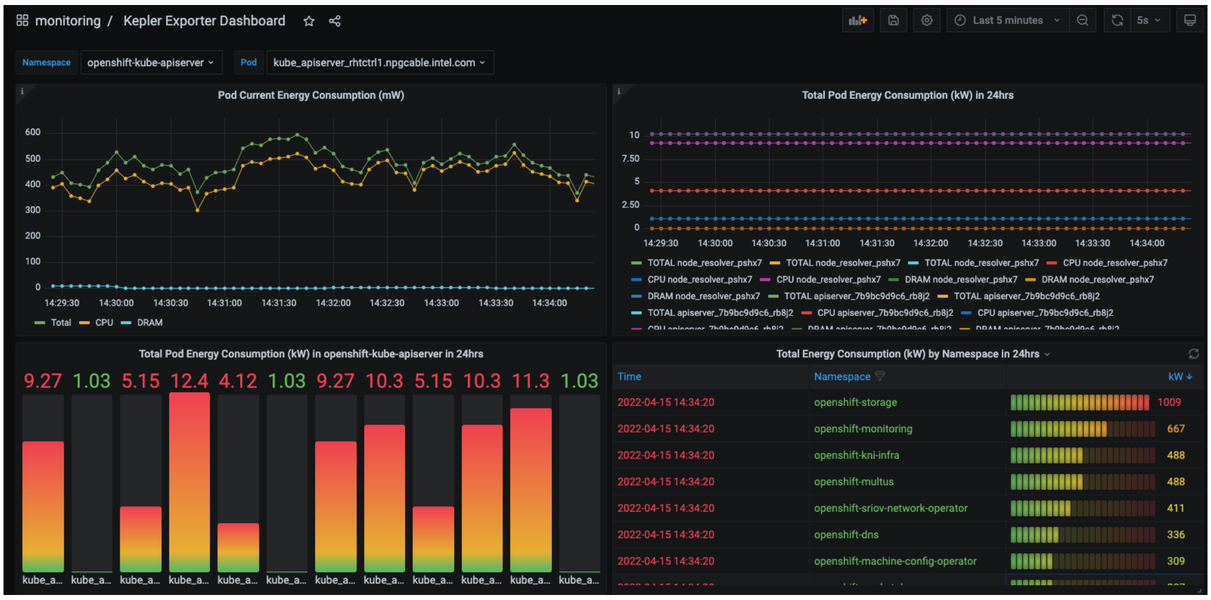
each processor component, pod in total, pod per each processor component), available input metrics, and model filters such as accuracy. In addition, the online trainer can be deployed as a sidecar container to the server (main container) to execute training pipelines and update the model on the fly when power metrics are available.

The default Kepler installation consists of a pod, the Kepler Exporter, deployed on each node of the Kubernetes cluster. The pod is then in charge of collecting all the metrics previously mentioned and interacting with the model server to get estimates if the actual metrics are not available.

Metrics can then be collected and stored using Prometheus and the OpenTelemetry standard for data visualization and post-processing. Examples of exported metrics are:

- **kepler_container_core_joules_total** (Counter) This measures the total energy consumption on CPU cores that a certain container has used.
- **kepler_container_gpu_joules_total** (Counter) This measures the total energy consumption on the GPUs that a certain container has used.
- **kepler_node_core_joules_total** (Counter) Similar to container metrics, it represents the aggregation of all containers running on the node and operating system (i.e., "system_process").
- **kepler_node_gpu_joules_total** (Counter) Similar to container metrics, it represents the aggregation of all containers running on the node and operating system (i.e., "system_process").

An example of a Grafana dashboard showing some of the metrics gathered by Kepler is shown in the picture below.



monitoring / Kepler Exporter Dashboard
Last 5 minutes
5s
Namespace
openshift-kube-apiserver
Pod
kube_apiserver_rhtctrl1.npgcable.intel.com
Pod Current Energy Consumption (mW)
Total
CPU
DRAM
Total Pod Energy Consumption (kW) in 24hrs
TOTAL node_resolver_pshx7
CPU node_resolver_pshx7
DRAM node_resolver_pshx7
TOTAL apiserver_7b9bc9d9c6_rb8j2
CPU apiserver_7b9bc9d9c6_rb8j2
Total Pod Energy Consumption (kW) in openshift-kube-apiserver in 24hrs
9.27 1.03 5.15 12.4 4.12 1.03 9.27 10.3 5.15 10.3 11.3 1.03
kube_a...

Total Energy Consumption (kW) by Namespace in 24hrs

Time	Namespace	kW
2022-04-15 14:34:20	openshift-storage	1009
2022-04-15 14:34:20	openshift-monitoring	667
2022-04-15 14:34:20	openshift-kni-infra	488
2022-04-15 14:34:20	openshift-multus	488
2022-04-15 14:34:20	openshift-sriov-network-operator	411
2022-04-15 14:34:20	openshift-dns	336
2022-04-15 14:34:20	openshift-machine-config-operator	309

Figure 12. Screenshot for Kepler metrics.

Funded by Horizon Europe Framework Programme of the European Union

5.3.2 Carbon-aware scheduling

Currently, small and on-premises datacenter are not required to be compliant with any "green" directives, hence their effort is mostly driven by the necessity to reduce economic costs; in some limited cases, energy-aware policies are implemented for the will to declare the operating company as compliant with the "green" economy.

For multiple reasons (often both economic and in terms of reputation), most hyperscalers already implement some energy-aware computing policies; we present here two possible examples, from Google and Microsoft.

5.3.2.1 Google's Carbon-aware Computing

Developed by Google for its data centers across the globe, this concept envisions shifting computation loads in time and space to take advantage of low-carbon electricity.

The core components are as follows:

- Fetching pipeline for next day's carbon intensity forecast.
- Power models pipeline for statistical model training which maps CPU usage to power consumption for various power domains characteristic to Google's heterogeneous infrastructure landscape.
- Load forecasting pipeline responsible for generating the next day's forecasts for workload demand (both flexible and inflexible) on the cluster level. An important characteristic of this pipeline is the evaluation of the forecasting error, which proves to be crucial for the following two components.
- Optimization pipeline co-optimizes the next day's expected carbon footprint and power peaks. The latter are subject to infrastructure and application service level objective (SLO) constraints as well as contractual and resource capacity limits.
- SLO violation detection raises a flag when the daily flexible demand of a cluster is not met. This triggers the cluster being excluded from shaping pool for a week in order for the load forecasts (component #3) to adapt to the rise in demand.

Funded by Horizon Europe Framework Programme of the European Union

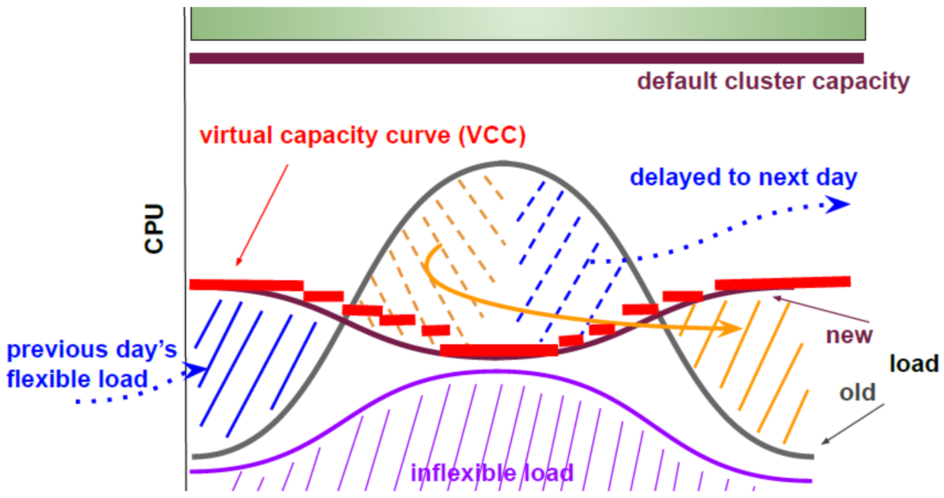


Figure 13. Demonstration of cluster load shaping by a VCC.

The algorithm yields the so-called Virtual Capacity Curves (VCCs), that are further used to reshape the peaks of flexible load and shift them towards more favorable times of the day.

5.3.2.2 Microsoft's Carbon-aware Kubernetes

With a similar purpose as Google, Microsoft puts forward the idea of expanding a vanilla Kubernetes scheduler with a weighting algorithm to optimize the node-selection process for the purposes of sustainability.

Apart from the fundamental components of Kubernetes and functionality developed by James et al. [83], this concept from Microsoft incorporates carbon intensity data and a customizable weighting algorithm. The most basic version of this algorithm deals with normalized "Marginal Operating Emission Rate" (MOER) for each cluster node. The idea is to take the MOER value of an individual node and divide it by the total MOER values across all nodes to get a normalized percentage weighting of each node. Optionally, a latency constraint, as well as predicted MOER values, can be built into the weighting algorithm for a more sophisticated decision process.

5.4 Intent Description Languages

FLUIDOS foresee the possibility to drive the virtual infrastructure through Intent-driven policies and languages, hence enabling to declare high-level goals (e.g., "deploy this application on multiple sites in France, at no more than 10ms from its end-user customers") instead of issuing imperative commands (e.g., "deploy this application on the Marseille-east site, on the Nice site, and more"). In this section we highlight the most relevant background for high-level intent-description languages.

Funded by Horizon Europe Framework Programme of the European Union

The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [89] is the de-facto modelling and representation paradigm for specifying Infrastructure as Code (IaC). TOSCA allows for the expression of topologies of cloud-based applications, their components and relationships, through node and relationship templates. These templates represent respectively the resources or components to be used in the deployment and the dependencies between the nodes [90]. TOSCA is an open standard designed to be cloud platform agnostic, unlike similar orchestration standards such as AWS CloudFormation or OpenStack Heat, which are tailored to their specific platforms. TOSCA compliant orchestrators such as xOpera [91] allow for cloud application deployment to arbitrary IaaS. In [92], Tamburri et al. elaborate on the possibility of intent modelling using TOSCA templates, and [93] describe a proof-of-concept Intent-based TOSCA Cloud Management Platform that transforms high-level intents, written in a natural-like language, into TOSCA YAML IaC definition files.

There exists a similar line of research in translating high level network intents into low-level network configurations that can be pushed and configure network devices. Intent Description Languages (IDLs) such as the Network Intent LanguagE (NILE) [94], [95] have been introduced. In [94], the "Lumi" chatbot uses NILE as an abstraction layer between the natural language description of the network intent provided by the end user and the final Merlin [96] network configuration language. Additionally, [97] converts multiple network intents described using NILE into a ".P4" configuration file. Natural Language Processing (NLP) techniques are used in [94], [98] to perform entity extraction, selection, and labelling, in conjunction with user feedback to correctly perform the translation of the intent into the intermediary representation.

MISSING SECTIONS

5.5 Resource and Service brokering

Resource and service brokering represent a key concept in FLUIDOS, at least in two directions:

- A small device, with limited resources, may need to give up some of its functions to a more complete FLUIDOS node, featuring full FLUIDOS capabilities, which takes care of its integration within a FLUIDOS environment.
- Cloud-native infrastructures, such as a data-center on-premises, may leverage some (optional) supporting component e.g., to discovery available resources and services among the many available clusters, and possibly to delegate to that component some of its complex operations (e.g., compare the most convenient offer for a given amount of desired resources).

Funded by Horizon Europe Framework Programme of the European Union

In this respect, a few solutions are currently available. The most appropriate is the Open Service Broker API[3] (OSBAPI), a specification for creating, managing, and consuming services in cloud-native environments. It defines a standard way for applications to discover, provision, and bind to services, regardless of the underlying infrastructure or platform. It allows for the creation of a marketplace of services that can be easily discovered and consumed by developers, and enables providers to build and offer services in a platform-agnostic manner. The Open Service Broker API works by providing a set of standard RESTful endpoints for managing services. These endpoints are designed to be called by a service broker client, which is typically integrated into a platform, such as a PaaS or a Kubernetes cluster.

The basic flow of the API is as follows:

- A developer requests a service through a platform-specific client, such as a command-line interface or a web UI.
- The client sends a request to the service broker, which acts as an intermediary between the platform and the service provider.
- The service broker communicates with the service provider to provision the requested service.
- Once the service is provisioned, the broker sends a response to the client, which includes the necessary credentials and connection information for the developer to access the service.
- The developer can then use the provided credentials to access the service and perform various operations, such as creating, reading, updating and deleting data.

The Open Service Broker API supports a variety of service operations, such as provisioning, binding, and deprovisioning, which can be used to create, configure, and manage services in a platform-agnostic way. It can be integrated into Kubernetes in several ways. One possible approach is to use the Kubernetes Service Catalog (k8s SIG)[4], which is a Kubernetes add-on that provides an implementation of the Open Service Broker API. The Service Catalog allows you to provision and manage services in a Kubernetes cluster by defining Custom Resource Definitions (CRDs) for services, plans, and bindings. Another approach is to use a Kubernetes Operator that is specifically designed to manage the lifecycle of services integrated with the Open Service Broker API. In summary, the Kubernetes Service Catalog provides an implementation of the Open Service Broker API, which makes it easy to integrate services into a Kubernetes cluster and manage them using Kubernetes constructs.

OSBAPI can be used to automate the provisioning and management of services such as databases, message queues, and other types of back-end services. It can also be used to

[3] https://www.openservicebrokerapi.org/

[4] https://github.com/kubernetes-retired/service-catalog.

Funded by Horizon Europe Framework Programme of the European Union

integrate services from different providers, allowing developers to easily switch between different service providers without having to change the code of their applications.

In summary, OSBAPI provides a standard way for cloud-native applications to discover, provision, and consume services, and to make it easy for developers to use and manage services in a platform-agnostic way (i.e., automation of service provisioning and management). However, it is limited to a single domain or cluster and may not include all the operations that a specific service provider may need. Moreover, it was made to deal with developers and to make them be able to automate DevOps operations inside the cluster where they are working on, allowing them to speed up the development process. Hence, it is not a real brokering system to deal with multiple domains as we would like to have in FLUIDOS; furthermore, it works on services level, meant as applications, and not on resource offering level (e.g., CPUs).

Finally, at the time of writing we should mention that the Service Catalog addon defined, developed and supported by the Kubernetes SIG, useful for the integration of OSBAPI with Kubernetes environments, has been archived in May 2022 and it is no longer supported by the community. At the same time the OSBAPI project is now only supported for CloudFoundry environments, hence limiting the applicability of this project.

The most notable pros and cons of the Open Service Broker API are listed below.

5.5.1 Pro and cons

Advantages

- Standardization: OSBAPI provides a standard way for applications to discover, provision, and consume services, regardless of the underlying infrastructure or platform. This allows for the creation of a marketplace of services that can be easily discovered and consumed by developers.
- Platform-agnostic: OSBAPI allows for the creation of services that can be consumed by multiple platforms, such as PaaS or Kubernetes clusters, without having to write platform-specific code.
- Automation: OSBAPI enables the automation of service provisioning and management, which can save time and reduce the complexity of managing services.
- Security and isolation: The single domain aspect of OSBAPI allows for better security and isolation of services and resources.

Limitations

- Limited to a single domain or cluster: Each instance of a service broker is scoped to a specific domain or cluster, and is only able to manage services within that domain or cluster. This can limit the ability to access services in other domains or clusters.

Funded by Horizon Europe Framework Programme of the European Union

- Limited to a specific set of operations: OSBAPI defines a set of standard operations for managing services, but it may not include all the operations that a specific service provider may need.
- Requires additional components: To use OSBAPI, additional components such as a service broker client and a service broker need to be integrated into the platform or cluster.

Funded by Horizon Europe Framework Programme of the European Union

6 ARCHITECTURE

6.1 TERMINOLOGY

This section presents the main concepts with respect to the terminology used in this project, and their relationship with respect to the vanilla Kubernetes terminology.

6.1.1 Kubernetes Node

A **Node** is a worker machine in Kubernetes, which may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by a control plane, which is (logically) unique across the entire Kubernetes clusters. A Node can host multiple pods, and the Kubernetes control plane automatically schedules pods on all available nodes, taking into account parameters such as the available resources (e.g., RAM, CPU) on each node, the availability of an already cached image, and more.

Each Kubernetes Node runs at least two components:

- **Kubelet**: process responsible for communication between the (centralized) Kubernetes control plane and the node itself; it manages the lifecycle of all the pods running on a machine.
- **Container runtime** (e.g., Docker): component responsible for pulling the container image from a registry, unpacking the container, and running it.

6.1.2 Kubernetes Cluster

A Kubernetes **Cluster** is a set of nodes that run containerized applications, which are made by different cooperating microservices. A Kubernetes cluster can be set up on different environments, with very different characteristics and resources, starting from managed clusters in a big datacenter, to small clusters featuring a few physical servers in a telco edge, or to tiny clusters that include a few resource-limited end-user devices.

A Kubernetes cluster consists of one master node and a number of worker nodes, ranging from a few devices, to some thousand servers. These nodes can either be physical computers or virtual machines, depending on the cluster. The master node controls the state of the cluster; for example, which microservices are running and their corresponding container images. The master node drives the scheduling process for all the running jobs, and it coordinates processes such as:

- Scheduling and scaling applications
- Maintaining a cluster's state
- Implementing updates

Funded by Horizon Europe Framework Programme of the European Union

6.1.3 FLUIDOS Node

A FLUIDOS **Node** is a **unique computing environment**, under the control of a **single administrative entity** (although different Nodes can be under the control of different administrative entities), composed of one or more machines and modeled with a common, extensible set of primitives that hide the underlying details (e.g., the physical topology), while maintaining the possibility to export the most significant distinctive features (e.g., the availability of specific services; peculiar HW capabilities). Overall, A FLUIDOS node is orchestrated by a single Kubernetes control plane, and it can be composed of either a **single device** or a **set of devices** (e.g., a datacenter). Device homogeneity is desired in order to simplify the management (physical servers can be considered all equals, since they feature a similar amount of hardware resources), but it is not requested within a FLUIDOS node. In other words, a FLUIDOS node corresponds to a Kubernetes cluster.

A FLUIDOS node includes a set of resources (e.g., computing, storage, networking, accelerators), software services (e.g., ready-to-go applications) that can be either leveraged locally or shared with other nodes. Furthermore, a FLUIDOS node features **autonomous orchestration capabilities**, i.e., (1) it accepts workload requests, (2) it runs the requested jobs on the administered resources (e.g., the participating servers), if application requirements and system security policies are satisfied, and (3) it features a homogeneous set of policies when interacting with other nodes.

6.1.4 FLUIDOS Domain

A FLUIDOS **Domain** is the set of FLUIDOS Nodes belonging to the same administrative domain, which have established peering relationships, and in which one of the existing FLUIDOS nodes is selected as master, called FLUIDOS SuperNode. A FLUIDOS domain leverages the same interfaces of FLUIDOS nodes to expose their aggregated information, enabling hierarchical interactions among different domains.

6.1.5 FLUIDOS SuperNode

A FLUIDOS Supernode is a FLUIDOS node that acts as a "master" for an entire FLUIDOS domain, becoming the entity that can establish peering relationships on behalf of all nodes of a FLUIDOS domain toward a third-party Node, and acting as "aggregator" of resources and services for all the entire FLUIDOS Domain.

6.1.6 FLUIDOS Cluster

A FLUIDOS **Cluster** is a **virtual space** spanning across a set of FLUIDOS nodes (also belonging to different administrative domains), in which all Kubernetes objects (e.g., pods, services, secrets, configmaps, etc.) are available such as in a traditional physical cluster. This enables running objects (e.g., microservices) to have direct access to the **portion** of computing,

Funded by Horizon Europe Framework Programme of the European Union

storage, network resources and services assigned to the virtual cluster by each participating FLUIDOS node, without boundaries (technical or administrative).

6.2 Interactions among FLUIDOS nodes

FLUIDOS nodes interact with each other according to two dimensions, horizontal and vertical, building a virtual continuum that can be completely decentralized, without any single point of control. In the following, we present an initial definition of the high-level concepts inherent with horizontal and vertical interactions, while the detailed aspects will be explored in WP3.

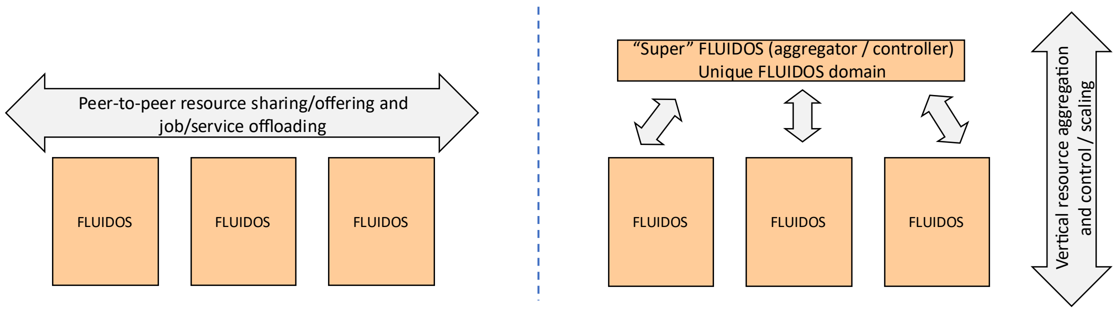

Peer-to-peer resource sharing/offering and job/service offloading
FLUIDOS
FLUIDOS
FLUIDOS
"Super" FLUIDOS (aggregator / controller) Unique FLUIDOS domain
FLUIDOS
FLUIDOS
FLUIDOS
Vertical resource aggregation and control / scaling

Figure 14. Horizontal and vertical interactions among FLUIDOS nodes.

6.2.1 Horizontal interactions

The horizontal (east-west) interaction enables the creation of a fluid domain among peers, which can share their resources and services, or part of them, based upon a set of policies (e.g., enable sharing from node X but not from node Y). Horizontal interactions are carried out according to a peer-to-peer paradigm, hence without the need for any centralized entity that controls and supervises the entire process. Applications started on a FLUIDOS node can leverage any resource, either local or remote, available in the new virtual space. The FLUIDOS orchestrator is responsible for determining the best location for each component based on (i) the features requested through the intent-based API (e.g., computing power, maximum latency between components and toward end users, resiliency properties, number and location of replicas, reduction of energy and costs or carbon, etc) and (ii) the additional policies set for the creation of the virtual space and/or the offloading of any component (e.g., no offloading on un-trusted domains for critical components). Each FLUIDOS node can dynamically create its own fluid domain, with an unlimited number of peers, if they accept to peer and share resources; peering can be turned on/off at will, with minimal overhead.

The establishment of a peering relationship is a multi-step process, which mainly involves the following phases: (i) *discovery*, allowing to discover about the existence of other peering candidates, (ii) *negotiation*, enabling FLUIDOS nodes to issue requests for resources and/or services to other nodes, which in turn advertise their available capabilities, (iii) *reservation and contract signing*, formalizing the acceptance of an offer from a FLUIDOS node, and possibly

Funded by Horizon Europe Framework Programme of the European Union

agreeing on compensations, (iv) peering, establishing the virtual fabrics required to enable seamless workload offloading to remote nodes (e.g., concerning networking, storage, …), (v) *usage*, leveraging the resources and services acquired by other FLUIDOS nodes to satisfy service requests and (vi) *depeering*, tearing down the previously established computing continuum abstractions, which are no longer necessary. The different steps briefly outlined above will be analyzed in a greater detail when discussing the different FLUIDOS components.

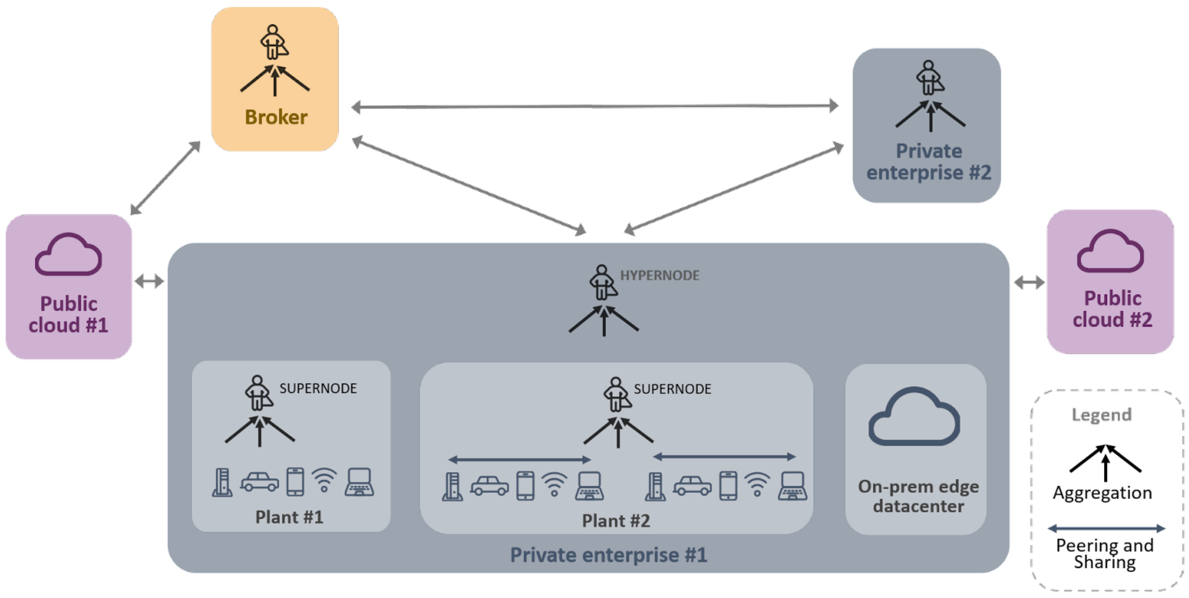


Figure 15. Horizontal and vertical interactions in FLUIDOS.

6.2.2 Vertical Interactions

The vertical (north/south) interaction introduces new concepts such as aggregation and hierarchical scaling into the picture. A fluid domain can be created by a FLUIDOS "supernode" that aggregates multiple nodes, hence exporting a virtual space that is the union of the resources (and services, objects) of the composing nodes.[5] The supernode-backed fluid domain is created with the same mechanisms already in place for the horizontal interactions; individual nodes are still in control of their own resources (ownership principle) and can withdraw them (if allowed by the signed contract). In addition, a special "tethered mode", available upon explicit configuration, allows individual nodes to be completely controlled by the supernode, giving up their local intelligence (and thus autonomy and ownership) in exchange for a simplified control plane. This mechanism, for instance, would be used to bring highly resource-constrained devices, such as micro-controllers, into the FLUIDOS ecosystem.

[5] Being an aggregation of multiple FLUIDOS nodes, a supernode possibly comprises a heterogeneous environment, under the control of multiple administrative entities. Still, it exposes the standard FLUIDOS node interfaces to hide these differences, presenting an abstract and aggregated view of the available resources and services, while seamlessly taking care of propagating workload execution to the appropriate FLUIDOS node, depending on the intent-driven specifications and additional policies.

Funded by Horizon Europe Framework Programme of the European Union

The aggregation paradigm can be applied recursively, with "supernodes" becoming part of a bigger "hyper-fluid domain", managed by an "hypernode" (or controlled, if the "tethered mode" is enabled), for a theoretically endless number of hierarchical levels. This provides the foundation for the hierarchical scaling property of FLUIDOS (recursive hierarchical architecture, from a single device to domains of domains). In this respect, the north/south interface, which enables interactions with different nodes (either local devices, clusters or other FLUIDOS "supernodes") all featuring the same interface, will be backed by novel resource aggregation algorithms that can aggregate the resources of each single node (normal, super, etc) in a scalable way. A possible example is shown in the figure above, with an enterprise sharing and aggregating at multiple levels within its ICT infrastructure, while establishing three different (direct) peering with external cloud providers and business partners.

Finally, a FLUIDOS Broker is an aggregation, consolidation and brokering (i.e., reselling) point that supports also interactions between multiple administrative domains, whose duties include the capability to (1) observe the resources offered by many domains and the associated performance when involved in task offloading; (2) to monitor (and predict) the quality of network connections; (3) to suggest the "best node" when asked for an offloading request. However, given that super/hyper nodes include brokering activities albeit limited within the boundaries of a single admin domain, FLUIDOS will reuse for the above nodes the great part of algorithms (and software) developed for the Broker, assigning to this component a major role in the architecture.

6.3 The FLUIDOS Software Stack

The FLUIDOS software stack is composed of four main layers, graphically depicted in Figure 16.

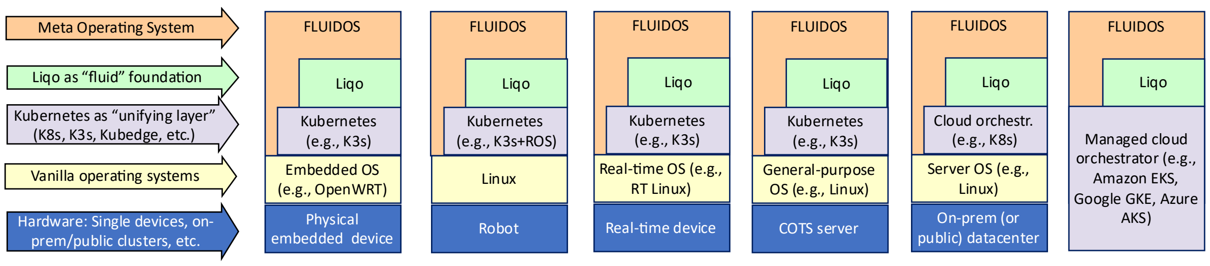

Meta Operating System
Liqo as "fluid" foundation
Kubernetes as "unifying layer" (K8s, K3s, Kubedge, etc.)
Vanilla operating systems
Hardware: Single devices, on-prem/public clusters, etc.
FLUIDOS
Liqo
Kubernetes (e.g., K3s)
Embedded OS (e.g., OpenWRT)
Physical embedded device
FLUIDOS
Liqo
Kubernetes (e.g., K3s+ROS)
Linux
Robot
FLUIDOS
Liqo
Kubernetes (e.g., K3s)
Real-time OS (e.g., RT Linux)
Real-time device
FLUIDOS
Liqo
Kubernetes (e.g., K3s)
General-purpose OS (e.g., Linux)
COTS server
FLUIDOS
Liqo
Cloud orchestr. (e.g., K8s)
Server OS (e.g., Linux)
On-prem (or public) datacenter
FLUIDOS
Liqo
Managed cloud orchestrator (e.g., Amazon EKS, Google GKE, Azure AKS)

Figure 16. The FLUIDOS software stack.

- Vanilla operating systems, which abstract the underlying hardware capabilities. There are strong technical and cultural reasons for having multiple operating systems on real devices, although many of them are Linux-based: necessity to support a given set of features (e.g., real-time processing), widespread adoption within a given community (e.g., Container OS),

Funded by Horizon Europe Framework Programme of the European Union

hardware requirements (e.g., low-cost devices), user constraints (e.g., usability in consumer-oriented electronics or smartphones).

- Kubernetes, which introduces a uniform layer on top of different infrastructures, regardless of whether they are end-user devices or larger cloud/edge data centers. Accounting for heterogeneous characteristics, Kubernetes distributions are available in different flavors: full-fledged Kubernetes (i.e., K8s) for small/large DCs, other distributions (e.g., Microk8s, K3s, KubeEdge) that target individual devices or small swarms of devices at the edge of the network. Kubernetes provides the minimum common denominator for FLUIDOS to build its services upon, leveraging its user-oriented primitives (e.g., replicas, deployments, stateful sets, services, etc.) that enable the deployment of user applications independently of the current distribution (e.g., K8s vs K3s), the size of the DC/node, and other characteristics (e.g., CPU architecture, i.e., Intel vs. ARM).
- Liqo, which brings in a multi-cluster abstraction on top of Kubernetes, enabling seamless offloading of workloads from one cluster to another. At the same time, it handles all the additional aspects required to make this process transparent from both the users and the applications point of view, including resource negotiation, cross-cluster network fabric setup, and synchronization of the appropriate artifacts. Overall, Liqo provides the foundation to enable the "resource virtualization" layer, exposing at the same time appropriate extension hooks to further enrich it with domain specific capabilities.
- FLUIDOS, which implements the full Meta Operating System capabilities. It builds on top, and it leverages the extension hooks made available by the underlying layers to enable the most advanced and domain specific capabilities.

The overall software stack shall acknowledge and support the possible usage of additional middleware frameworks widely adopted in certain communities, such as ROS and MQTT.

6.4 FLUIDOS NODE ARCHITECTURE

A FLUIDOS node builds on top of Kubernetes, which takes care of abstracting the underlying (physical) resources and capabilities in a uniform way, no matter whether dealing with single devices or full-fledged clusters (and the actual operating system) while providing at the same time standard interfaces for their consumption. Specifically, it properly extends Kubernetes with new control logic responsible for handling the different node to node interactions, as well as to enable the specification of advanced policies and intents (e.g., to constrain application execution), which are currently not understood by the orchestrator.

Given this precondition, the main architectural components of a FLUIDOS node are depicted in Figure 17, and converge around the Node Orchestrator and the Available Resources database. The former is in charge of orchestrating service requests, either on the local node or on remote nodes of the same fluid domain, coordinate all the interactions with local components (e.g., local scheduler) and remote nodes (e.g., to set up the computing / network / storage / service fabrics), and make sure that the service behaves as expected (e.g., honoring

Funded by Horizon Europe Framework Programme of the European Union

trust and security relationships). The latter keeps up-to-date information about resources and services available either locally or acquired from remote nodes, following the resource negotiation and acquisition process. Additional modules (and their companion communication interfaces), that are described in the following sub-sections, are required to handle the discovery of other FLUIDOS nodes and carry out the resource negotiation process, to monitor the state of the virtual infrastructure and to make sure that offloaded workloads/services behave as expected both in terms of security and negotiated SLAs, to take care of security and privacy issues (e.g., isolation), and to create the virtual continuum within the fluid space. The tasks handled by each module are detailed in the following.

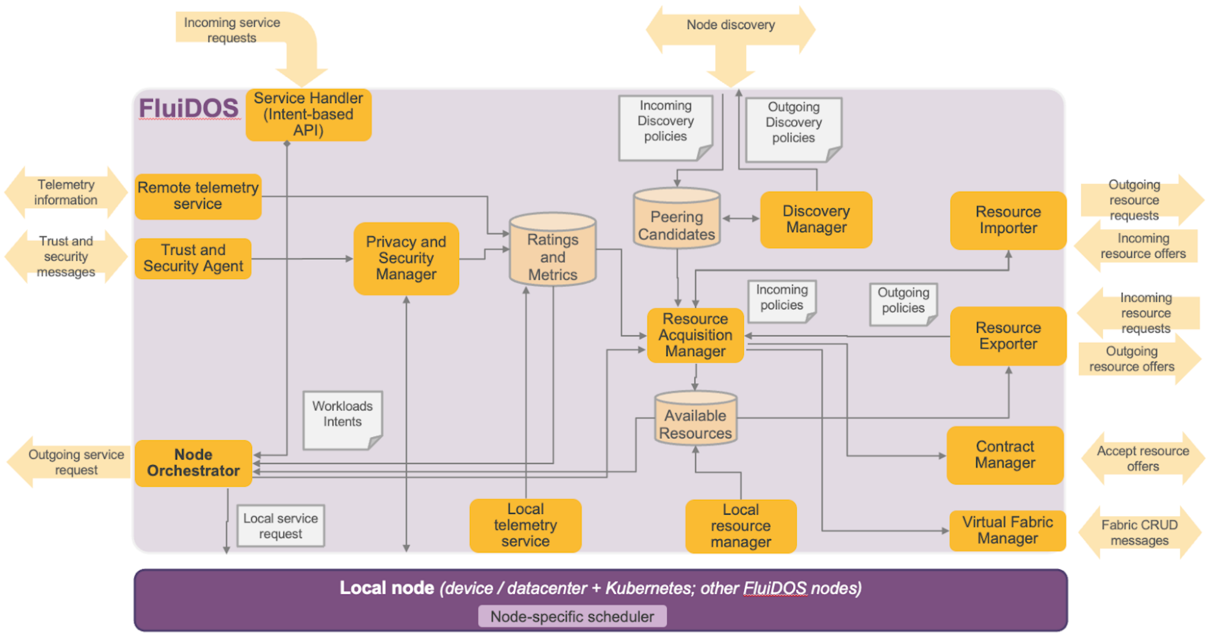


Figure 17. FLUIDOS logical architecture (see Appendix for full-size representation).

6.4.1 Discovery Manager

The *discovery manager* is the module responsible for the discovery of other FLUIDOS nodes, producing as output a local database of feasible *peering candidates*. Specifically, each peering candidate is characterized by a globally unique identifier, the set of parameters necessary during the peering and resource acquisition phase (e.g., target network endpoints, …), as well as a set of distinguishing features (e.g., geographical location, whether it is available to sell computing resources, specific hardware functionalities or software services), and possibly including pricing/billing models. These features are expected to be exposed at a high level (e.g., through generic key/value labels); they enable both an initial policy driven filtering (i.e., excluding undesired nodes from the list of *peering candidates*) and a priori filtering and ranking during the resource acquisition phase, while leaving that phase to deal with the negotiation in terms of quantities, cost, and more detailed aspects. More specifically, filtering and ranking might be based on different parameters, either based on previous experience or attached to the node during discovery, including reputation, availability to offer

Funded by Horizon Europe Framework Programme of the European Union

certain categories of resources and services, and more (see Appendix B for a complete example of Discovery Workflow).

Multiple discovery approaches are foreseen to account for different requirements, including and not limited to:

- FLUIDOS catalogs, i.e., directories of FLUIDOS nodes (a catalog might be public or require proper authentication to be queried/joined). Each node can subscribe to multiple catalogs, hence discovering about all other nodes therein listed, as well as announce itself in multiple catalogs. Additionally, catalogs possibly represent trustworthy anchors to gather and expose information concerning the reputation of a given node in the specific domain, based on its historical behavior and the feedback received from its peers.
- Multicast DNS, enabling seamless on-LAN clustering of independent devices.
- Manual configuration, specifying the appropriate parameters of the target peering candidate, as a fallback approach if none of the others is suitable in the given scenario.

The entire process is driven by a set of user-specified policies, which can be divided in:

- Outgoing: specify whether the current node should be announced by means of a certain approach or not (e.g., only through a subset of the known catalogs).
- Incoming: filter out the peers announced through a subset of approaches, those that do not expose the desired capabilities (e.g., I'm only interested in European nodes available to sell computing resources), or do not match the desired trust level (based on past experience and peers' ratings, as computed by the *privacy and security manager*).

The overall discovery process is summarized in the figure below.

Funded by Horizon Europe Framework Programme of the European Union

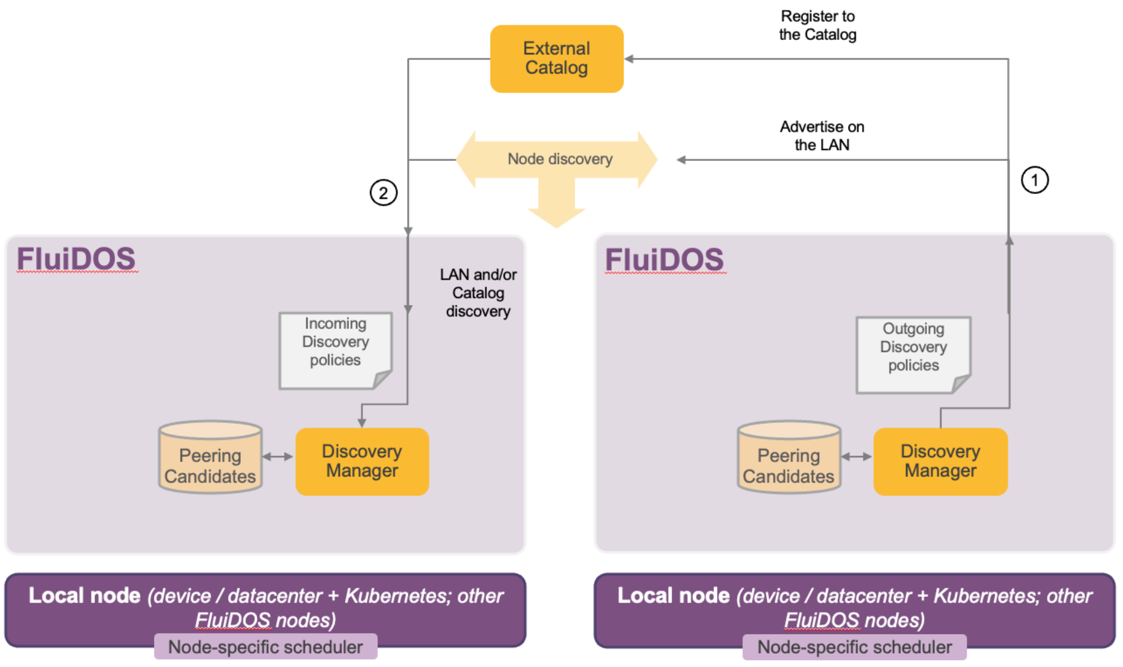

Register to the Catalog
External Catalog
Advertise on the LAN
Node discovery
2
1
FluiDOS
LAN and/or Catalog discovery
Incoming Discovery policies
Peering Candidates
Discovery Manager
FluiDOS
Outgoing Discovery policies
Peering Candidates
Discovery Manager
Local node (device / datacenter + Kubernetes; other FluiDOS nodes)
Node-specific scheduler
Local node (device / datacenter + Kubernetes; other FluiDOS nodes)
Node-specific scheduler

Figure 18. FLUIDOS discovery process.

Funded by Horizon Europe Framework Programme of the European Union

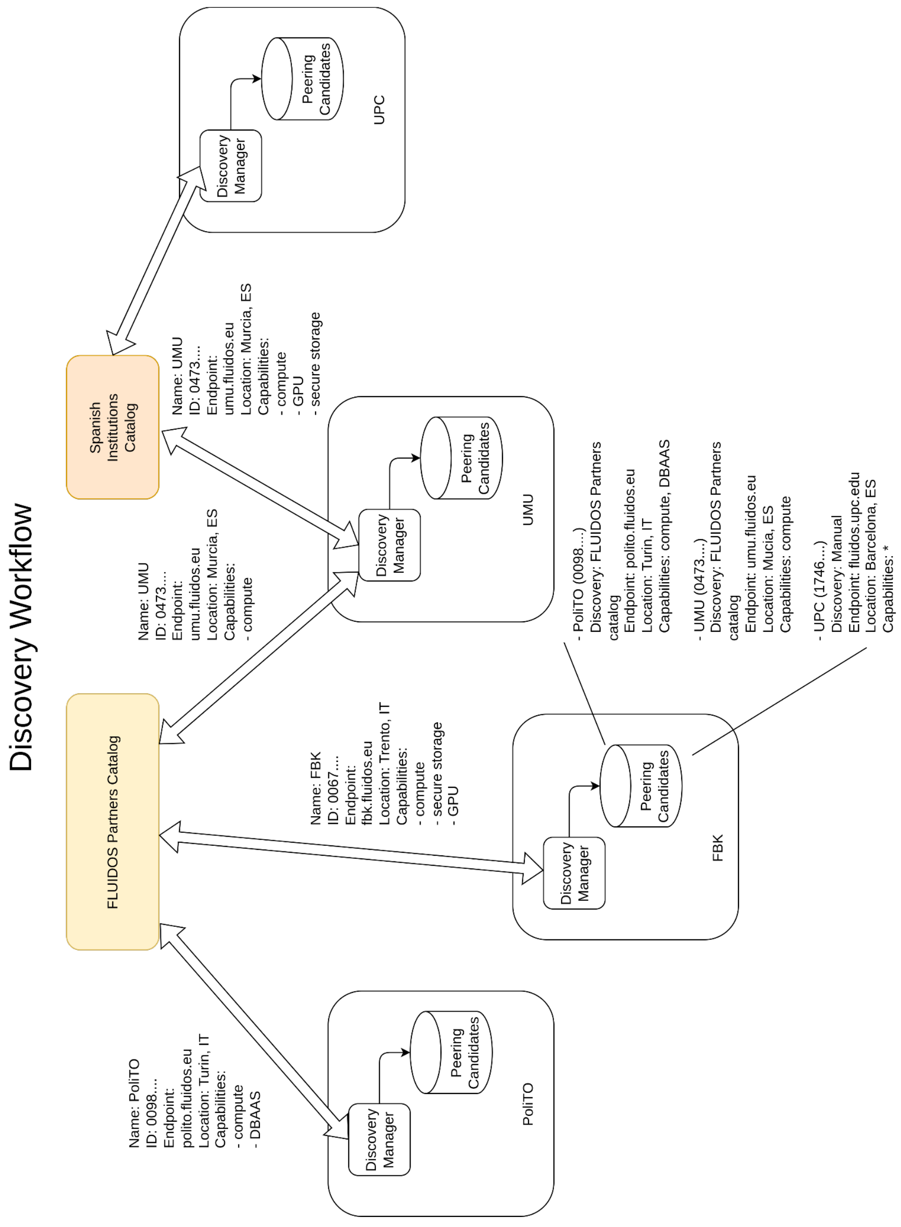


Figure 19. FLUIDOS discovery workflow.

6.4.2 Node Orchestrator

The *node orchestrator* is the FLUIDOS module responsible for the orchestration of the service requests, either on the local node, or offloading them to a remote FLUIDOS node, based on the current snapshot of the *available resources* database. Additionally, it interacts with and coordinates other components, mainly the *resource acquisition* manager, to trigger the acquisition of new resources and the setup of the appropriate network and storage fabrics enabling transparent execution continuum, in case already available ones are not sufficient to satisfy the incoming request.

Funded by Horizon Europe Framework Programme of the European Union

Service requests are specified, either by the DevOps users or propagated by another FLUIDOS node, through an intent-driven API; i.e., describing what the result should be, rather than how it should be achieved, hence facilitating service composition and QoS specification, while hiding at the same time low level details. Hence, each request is enriched with a set of soft (i.e., desired) and hard (i.e., mandatory) constraints expressed in terms of high-level service characteristics (e.g., geographical location, maximum latency, need for specialized hardware, HA guarantees, cost, …), rather than low-level implementation details (e.g., manually constrain the workloads to a specific subset of servers). Soft constraints are associated with priorities, to disambiguate possible conflicting requirements, and favor the most relevant ones (e.g., in case reducing latency implies at the same time an increase in the execution costs, and both should be minimized). It is worth mentioning that service characteristics involve both aspects directly under the control of a FLUIDOS node (e.g., the availability of certain hardware and its performance), either local or remote, and of their interconnection (e.g., concerning the network). The telemetry services are responsible for retrieving the information required to convert intents into actual scheduling decisions. The specification of the intent driven API, as well as the decision about the scheduling granularity (i.e., per-application or per-microservice) will be carried out as part of WP4.

Once a service request is received, the *node orchestrator* takes care of selecting the target FLUIDOS node for its execution (possibly the current node itself), based on the specified constraints. To this end, it reads the *available resources* database, which enumerates the list of currently available resources, and their characteristics. This database is fed by the *local resource manager*, which lists the resources (and services) exposed by the node itself (in the FLUIDOS sense, it can correspond to an entire Kubernetes cluster), as well as contains those previously acquired from other nodes, as discussed in the *resource acquisition manager* section. This list is possibly additionally filtered based on past experience reports, as contained in the *ratings and metrics* database. The definition of an appropriate scheduling algorithm will be part of WP4, while additional, domain-specific algorithms will be developed in the corresponding WP (e.g., WP6 for an energy-aware solution) (see Appendix C for the complete service request workflow in case resources are already available).

In case no match is found, the *node orchestrator* triggers the *resource acquisition* module, which starts querying the list of *peering candidates* with the objective to acquire the necessary resources. Eventually, either the node orchestrator gives up, signaling the impossibility of finding a suitable FLUIDOS node for the workload execution, or it outputs the target execution node. If it corresponds to a remote one, the service request is offloaded through the remote service handler, recursively triggering the same orchestration process (a loop resolution algorithm is foreseen to prevent the occurrence of scheduling loops, causing the workload to never be scheduled on a real node, in case multiple recursion steps are allowed). Alternatively, the workload is fed to the local scheduler (either the vanilla Kubernetes one or an enriched version featuring domain-specific knowledge), delegating to it the selection of the target server it will be actually executed on (see Appendix D for the complete service request workflow in case resources are not yet available).

Funded by Horizon Europe
Framework Programme of the European Union

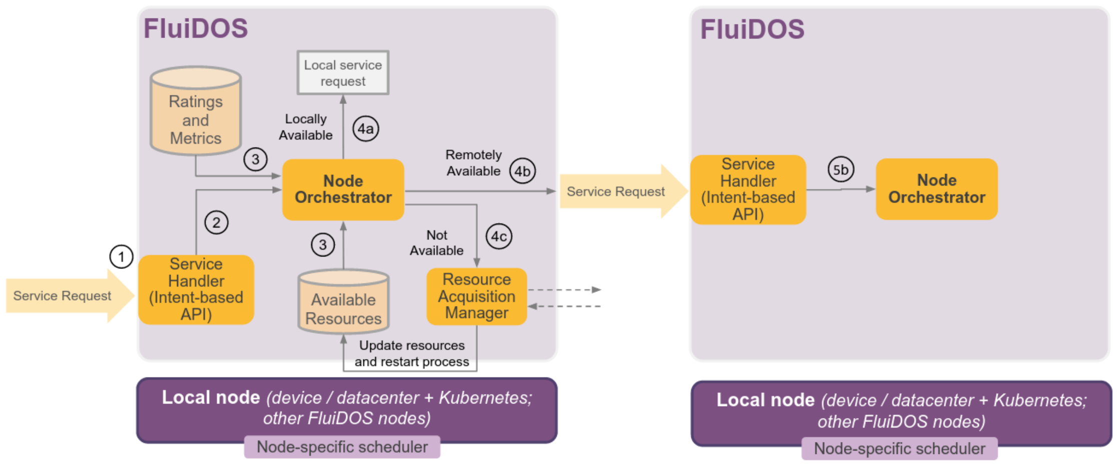

FluiDOS
Ratings and Metrics
Local service request
Locally Available
4a
3
Node Orchestrator
Remotely Available
4b
2
3
Not Available
4c
1
Service Request
Service Handler (Intent-based API)
Available Resources
Resource Acquisition Manager
Update resources and restart process
Local node (device / datacenter + Kubernetes; other FluiDOS nodes)
Node-specific scheduler
Service Request
FluiDOS
Service Handler (Intent-based API)
5b
Node Orchestrator
Local node (device / datacenter + Kubernetes; other FluiDOS nodes)
Node-specific scheduler

Figure 20. FLUIDOS service requests.

6.4.3 Resource Acquisition Manager

The *resource acquisition* manager is the FLUIDOS module responsible for the negotiation process performed to acquire resources and services from remote FLUIDOS nodes. It can be triggered either proactively, based on policies, to ensure that a given amount of resources (with certain characteristics) is always available to fulfill foreseen future requests, or by the node orchestrator, reacting to the lack of matching resources to satisfy a service request.

Whatever the case, the module reads the *peering candidates* database, appropriately filtered and ranked based on the service characteristics (e.g., keep only those offering resources in a certain country, sorted by a past experience-driven rating), and then sends a *resource request* (through the *resource importer* module) message to a shortlist of selected candidates, enumerating the amount of requested capabilities (e.g., 10 CPU cores, 25GB of RAM and a TPM) and possibly a set of related intent-driven characterizing constraints. New resources might be requested regardless of whether the target node is already offering some resources or not.

Resource requests are received by the *resource exporter* module on the counterpart cluster, which elaborates them and, based on the content of the *available resources database* and user-defined policies, decides whether to formulate a corresponding *resource offer* or deny the request (e.g., since it cannot be satisfied, or due to a lack of trust). A *resource offer* might exactly match the request, include only a subset of the requested resources (e.g., based on current availability, or due to a limit in the maximum amount that can be offered to remote nodes), or include a superset of the requested resources (e.g., since they are only offered in pre-defined size blocks). Additionally, it specifies the cost, as well as additional constraints associated with the offered resources (e.g., whether they have been reserved until acceptance, and for how much time).

Funded by Horizon Europe Framework Programme of the European Union

Different business models might be adopted to fulfill different contexts; for the sake of exemplification, we consider here a reserved, and a *pay-per-use* approach. In the former, upon acceptance, the negotiated resources are reserved, thus billed, to a given FLUIDOS node (regardless of whether they are consumed). Alternatively, the pay-per-use model enables higher dynamism, with a node paying only the resources effectively used in each moment, although without the guarantee of their continuous availability (e.g., due to resource overcommitment, and a temporary burst of requests by multiple nodes). It is worth mentioning that, in hierarchical scenarios (i.e., involving supernodes and hypernodes), a recursive approach might be desired, thus allowing the target cluster to trigger in turn the *resource acquisition manager*, attempting to acquire the resources to be offered from other clusters, and effectively playing the role of a reseller. Yet, in this case, a loop resolution algorithm is of utmost importance to prevent the possibility of (even indirectly) attempting to acquire resources from a FLUIDOS node already part of the chain.

At this point, the originating *resource acquisition manager* collects (through the resource *importer* module) the resulting set of *resource offers*, it filters and ranks them based on user-defined policies (e.g., prefer the ones exactly matching the request, exclude those costing more than a given threshold, …), eventually deciding which ones to accept. Then, the contract manager module has to confirm the acceptance of the selected resource offers, possibly leveraging smart contracts to formalize the exchange in terms of money and resources/services, and the acquired resources are added to the *available resources database*. If no matching offer is received, the process can optionally continue sending new requests to the next subset of *peering candidates*, until the list has been completely explored.

The resource acquisition process is summarized in the figure below, whereas the complete resource acquisition workflow is in the Appendix E.

Funded by Horizon Europe
Framework Programme of the European Union

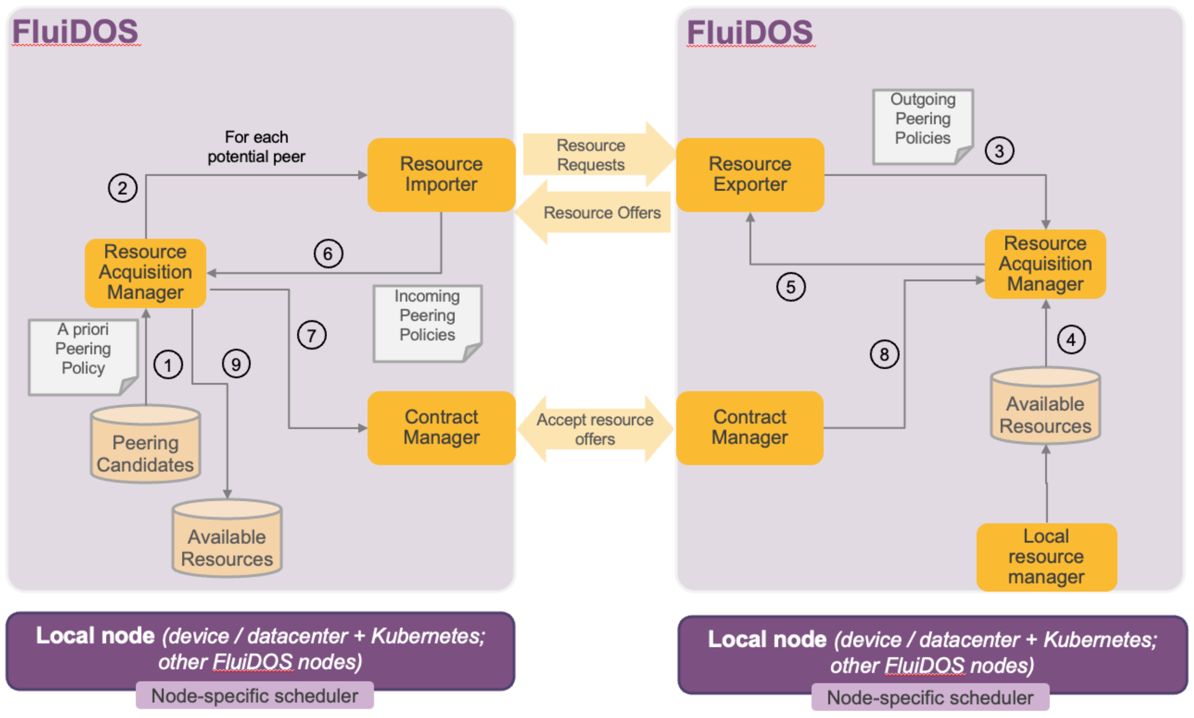


Figure 21. FLUIDOS resource acquisition process.

Funded by Horizon Europe Framework Programme of the European Union

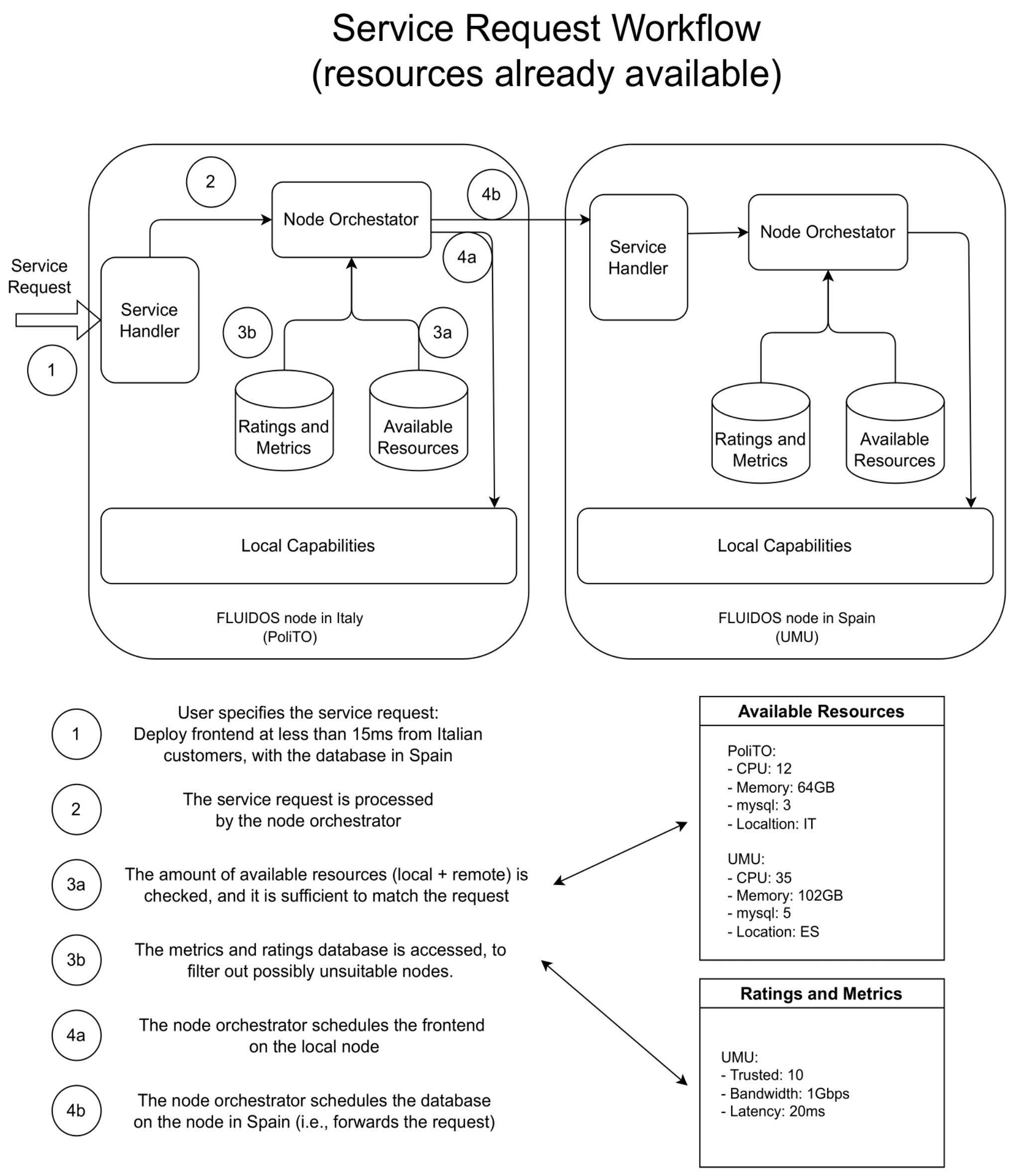
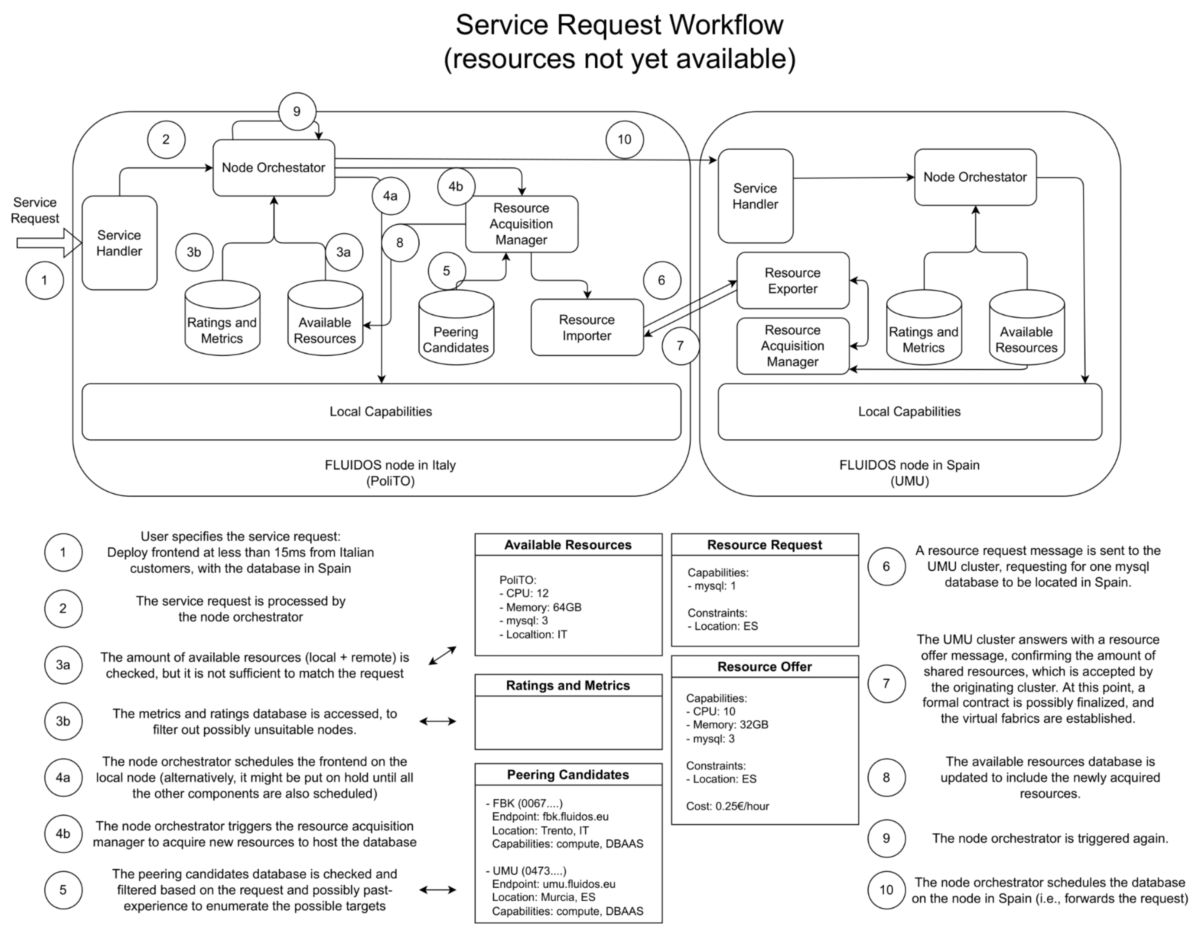

Service Request Workflow
(resources already available)
Service Request
1
Service Handler
2
Node Orchestator
3b
3a
4a
4b
Ratings and Metrics
Available Resources
Local Capabilities
FLUIDOS node in Italy (PoliTO)
Service Handler
Node Orchestator
Ratings and Metrics
Available Resources
Local Capabilities
FLUIDOS node in Spain (UMU)
1 User specifies the service request: Deploy frontend at less than 15ms from Italian customers, with the database in Spain
2 The service request is processed by the node orchestrator
3a The amount of available resources (local + remote) is checked, and it is sufficient to match the request
3b The metrics and ratings database is accessed, to filter out possibly unsuitable nodes.
4a The node orchestrator schedules the frontend on the local node
4b The node orchestrator schedules the database on the node in Spain (i.e., forwards the request)
Available Resources
PoliTO: - CPU: 12 - Memory: 64GB - mysql: 3 - Localtion: IT
UMU: - CPU: 35 - Memory: 102GB - mysql: 5 - Location: ES
Ratings and Metrics
UMU: - Trusted: 10 - Bandwidth: 1Gbps - Latency: 20ms

Figure 22. FLUIDOS Service Request workflow (resources already available).

Funded by Horizon Europe Framework Programme of the European Union



1 User specifies the service request: Deploy frontend at less than 15ms from Italian customers, with the database in Spain

2 The service request is processed by the node orchestrator

3a The amount of available resources (local + remote) is checked, but it is not sufficient to match the request

3b The metrics and ratings database is accessed, to filter out possibly unsuitable nodes.

4a The node orchestrator schedules the frontend on the local node (alternatively, it might be put on hold until all the other components are also scheduled)

4b The node orchestrator triggers the resource acquisition manager to acquire new resources to host the database

5 The peering candidates database is checked and filtered based on the request and possibly past-experience to enumerate the possible targets

Available Resources

PoliTO:
- CPU: 12
- Memory: 64GB
- mysql: 3
- Localtion: IT

Ratings and Metrics

Peering Candidates

- FBK (0067....)
  Endpoint: fbk.fluidos.eu
  Location: Trento, IT
  Capabilities: compute, DBAAS
- UMU (0473....)
  Endpoint: umu.fluidos.eu
  Location: Murcia, ES
  Capabilities: compute, DBAAS

Resource Request

Capabilities:
- mysql: 1

Constraints:
- Location: ES

Resource Offer

Capabilities:
- CPU: 10
- Memory: 32GB
- mysql: 3

Constraints:
- Location: ES

Cost: 0.25€/hour

6 A resource request message is sent to the UMU cluster, requesting for one mysql database to be located in Spain.

7 The UMU cluster answers with a resource offer message, confirming the amount of shared resources, which is accepted by the originating cluster. At this point, a formal contract is possibly finalized, and the virtual fabrics are established.

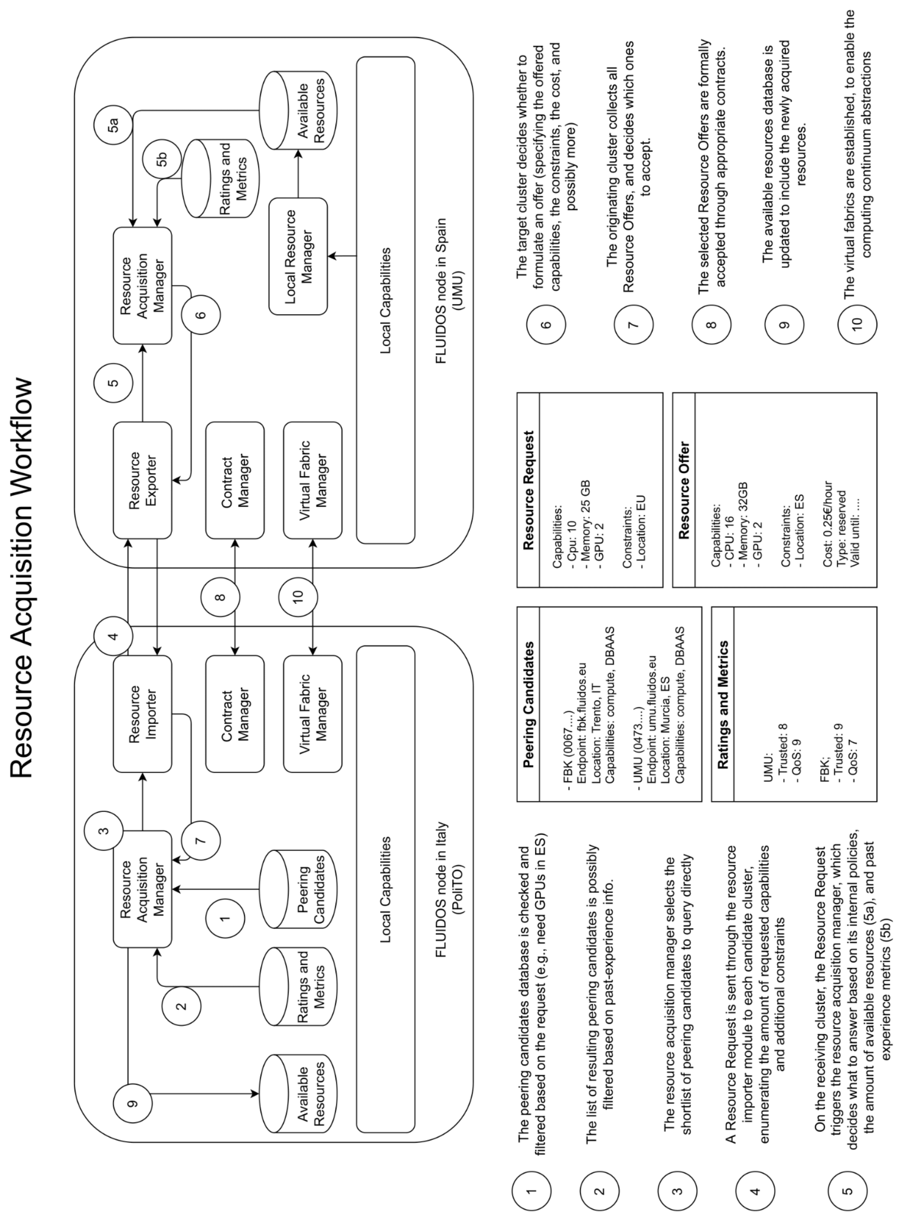
8 The available resources database is updated to include the newly acquired resources.

9 The node orchestrator is triggered again.

10 The node orchestrator schedules the database on the node in Spain (i.e., forwards the request)

Figure 23. FLUIDOS Service Request workflow (resources not yet available).

Funded by Horizon Europe Framework Programme of the European Union



Peering Candidates

- FBK (0067....)
  Endpoint: fbk.fluidos.eu
  Location: Trento, IT
  Capabilities: compute, DBAAS
- UMU (0473....)
  Endpoint: umu.fluidos.eu
  Location: Murcia, ES
  Capabilities: compute, DBAAS

Ratings and Metrics

UMU:
- Trusted: 8
- QoS: 9

FBK:
- Trusted: 9
- QoS: 7

Resource Request

Capabilities:
- Cpu: 10
- Memory: 25 GB
- GPU: 2

Constraints:
- Location: EU

Resource Offer

Capabilities:
- CPU: 16
- Memory: 32GB
- GPU: 2

Constraints:
- Location: ES

Cost: 0.25€/hour
Type: reserved
Valid until: ....

1 The peering candidates database is checked and filtered based on the request (e.g., need GPUs in ES)

2 The list of resulting peering candidates is possibly filtered based on past-experience info.

3 The resource acquisition manager selects the shortlist of peering candidates to query directly

4 A Resource Request is sent through the resource importer module to each candidate cluster, enumerating the amount of requested capabilities and additional constraints

5 On the receiving cluster, the Resource Request triggers the resource acquisition manager, which decides what to answer based on its internal policies, the amount of available resources (5a), and past experience metrics (5b)

6 The target cluster decides whether to formulate an offer (specifying the offered capabilities, the constraints, the cost, and possibly more)

7 The originating cluster collects all Resource Offers, and decides which ones to accept.

8 The selected Resource Offers are formally accepted through appropriate contracts.

9 The available resources database is updated to include the newly acquired resources.

10 The virtual fabrics are established, to enable the computing continuum abstractions

Figure 24. FLUIDOS Resource Acquisition workflow.

6.4.4 Virtual Fabric Manager

The *virtual fabric manager* is the FLUIDOS component in charge of establishing the computing continuum abstractions to enable the seamless execution of workloads spread across multiple nodes. Specifically, it takes action once a resource offer is accepted by the *resource acquisition manager*, and appropriately sets up the virtual node abstraction, along with the network and storage fabrics through the interaction with the remote node. Symmetrically, it takes care of tearing down the virtual fabric once resources are no longer acquired from a given node. This task is fulfilled by Liqo, a project which extends the Kubernetes abstractions to multi-cluster scenarios, considering three main directions:

Funded by Horizon Europe Framework Programme of the European Union

- Virtual node: abstracting a remote cluster (i.e., FLUIDOS node) as a local Kubernetes node, performing at the same time the synchronization of the necessary artifacts (e.g., configurations, service endpoints, …) to enable the seamless remote execution of unmodified workloads.
- Network fabric: ensuring that correlated workloads spread across multiple FLUIDOS nodes can transparently and securely communicate among each other, as if they were located on the same hosting node. The network fabric automatically manages the possible conflicts (e.g., in terms of overlapping addressing spaces) which inevitably arise in highly dynamic and uncoordinated scenarios.
- Storage fabric: enabling a data continuum to allow workloads to attach data lakes on the hosting FLUIDOS node. Following the data gravity approach, it prevents the need for expensive migration of data across FLUIDOS nodes, as well as to comply with law regulations (e.g., GDPR), ensuring that data is only stored and accessed in the desired locations.

6.4.5 Privacy and Security Manager

The *privacy and security manager* is the FLUIDOS module, investigated and detailed in WP5, in charge of guaranteeing the security of the different parties involved in the resource continuum, and it is intertwined with all the other different processes. This task is additionally fulfilled through the interaction with the *trust and security agent*, a trust anchor part of each FLUIDOS node that certifies the correctness of predefined operations.

Node discovery: it ensures with appropriate proof mechanisms the trustworthiness of the distinguishing features advertised by remote nodes, so that they can be reliably leveraged upon for filtering and ranking purposes. Each FLUIDOS node is also possibly associated with a reputation value, built from the collective past experience of the entire continuum components, and based on its behavior (e.g., in terms of workloads execution), as well as the adherence to the offered capabilities.

Resource acquisition: the *rating and metrics database* fed by this module is leveraged to filter and rank the list of feasible *peering candidates* based on previous experience, both collected directly by the local node and referring to a global reputation value. Additionally, it plays a role in proving that the offered capabilities are actually available on the host node.

With respect to the execution of the remote workload, the Privacy and Security Manager must guarantee the following properties:

- **Remote workload execution (host perspective)**: the hosting FLUIDOS node is guaranteed that offloaded workloads do not harm the local system, nor they attempt to perform malicious actions towards external domains. This is achieved by appropriately confining one or more related workloads in a dedicated sandbox (e.g., a container), thus configuring the appropriate limitations in terms of resource consumption (i.e., to prevent using resources without a previous agreement),

Funded by Horizon Europe Framework Programme of the European Union

operations that can be performed (e.g., system call subsets that can be invoked), and network connectivity (i.e., forbidding hosted workloads from accessing unauthorized services). In addition, measures are taken to prevent side-channel attacks aimed at trying to infer confidential information about the containerized applications. Finally, it supervises the provisioning of additional mechanisms (e.g., TPM, secure storage, …) possibly agreed upon during the resource acquisition phase and requested by the current workloads through the intent-based API.

- **Remote workload execution (guest perspective)**: the guest FLUIDOS node should be provided with additional guarantees that offloaded workloads are being executed correctly (i.e., without being tampered with), as well as that the telemetry data forwarded by the hosting node is correct (e.g., about resources' usage). These pieces of information are then leveraged by this module to enrich the reputation value about the specific FLUIDOS node, stored into the ratings and metrics database, later queried during subsequent resource acquisition phases.

Finally, the Privacy and Security Manager is responsible for the dynamic selection and configuration of security services, what is usually called *Security Orchestration*. To this account, it interacts with the local orchestrator (e.g., to deploy decoys) and the telemetry service (to gather information about the status of the infrastructure and existing services).

6.4.6 Telemetry Service

The *telemetry service* is the FLUIDOS component responsible for the monitoring of the infrastructure (e.g., actual CPU load, memory, network, as well as possibly more detailed indexes such as memory page faults), including the collection of all the observability parameters key to enforce and verify the satisfaction of the workload requirements expressed through the intent-based API.

It consists of a *local telemetry service*, responsible for the performance of the local FLUIDOS node, which is additionally leveraged by the *local resource manager* to derive the locally available resources. The *remote telemetry service*, on the other hand, exposes to each peered node the aggregated information concerning the guest workloads therein hosted, thus enabling for the monitoring of their execution, and the verification that negotiated SLAs are respected. In this context, the remote *trust and security agent* plays a key role to prevent remote nodes from cheating about application performance. The outcome of the monitoring process enriches the *ratings and metrics database*, hence driving subsequent resource acquisition processes.

6.4.7 Cost Manager

The *Cost Manager* is the FLUIDOS module responsible for evaluating the burdens of carrying out a computational load on a node. Hereby, the burdens can be both monetary or non-monetary (in particular environmental). Three cost functions have been identified so far:

Funded by Horizon Europe Framework Programme of the European Union

- Operational monetary
- Environmental, in particular carbon emissions
- HW production monetary

However, the scheduler related to the monetary costs of HW production might be substituted with the results of a study on a new and more dynamic DC architecture that is being carried out by IBM. This new architecture could bring about monetary and environmental savings and therefore serve as an indirect optimization tool for HW-related costs. Contacts to the OpenFabrics Alliance within the consortium can bring synergies on this task.

Whether all three or just the first two of these cost functions will be implemented, it is certain that there will be more than a single cost type. The cost function thus needs to be generic and parameterizable.

The environmental scheduler will most likely be implemented among FLUIDOS nodes (i.e., Kubernetes Clusters), as otherwise working within Kubernetes internals could get quite challenging. The manager will nevertheless have access to cluster internals to compute and publish its cost function.

The cost manager is so far foreseen to synergize with the local resource manager. This is because the algorithm of the environmental scheduler must closely monitor the resources and sensors on the given machine as well as have access to the carbon intensity of the local electricity grid of that machine.

6.5 FLUIDOS COMPONENTS STATUS OVERVIEW

The figure below summarizes the status of the building blocks envisioned for the FLUIDOS architecture, highlighting those already available in Liqo, the ones that are present in an initial version, but require deep customization and extension to fulfill the FLUIDOS requirements, as well as the ones expected to be designed and implemented from scratch as part of FLUIDOS.

Funded by Horizon Europe Framework Programme of the European Union

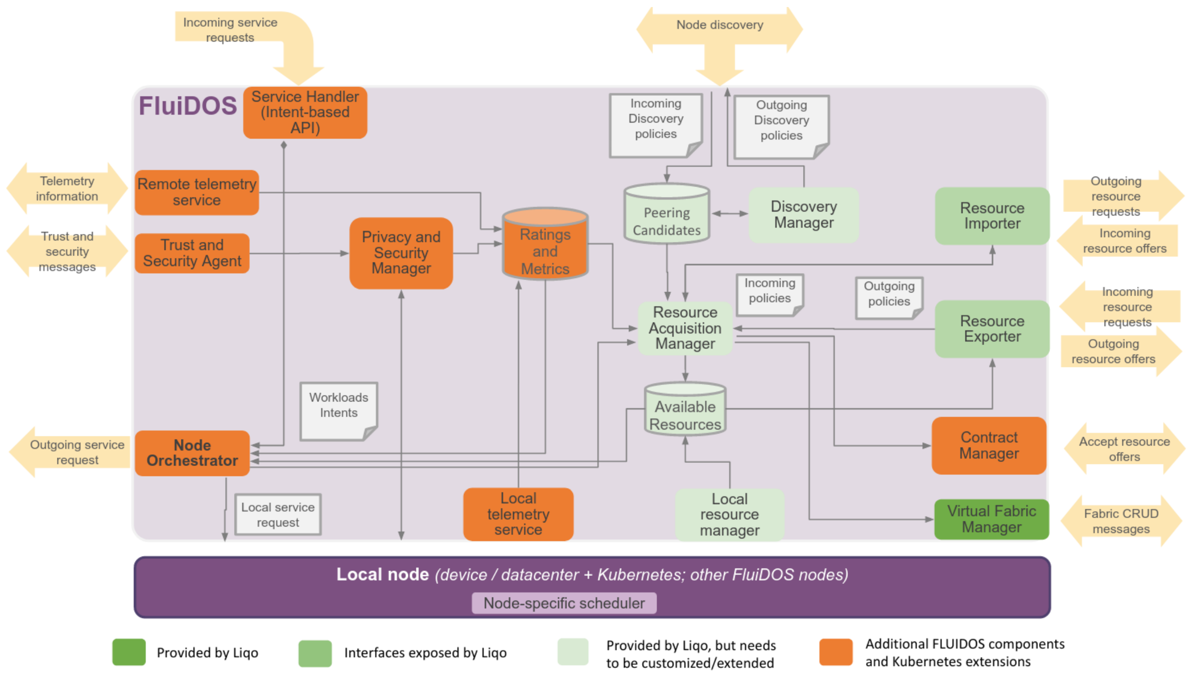


Figure 25. FLUIDOS component status overview.

Funded by Horizon Europe Framework Programme of the European Union

7 ADDITIONAL COMPONENTS: CATALOG & BROKER

While the architecture defined in the previous Section is enough to build, maintain and operate a FLUIDOS domain, some new components (namely, catalogs, brokers) may be helpful to increase the value of the FLUIDOS solution, as well as adding new functionalities and/or (optionally) simplifying some of its operations in some given conditions.

This section presents the main concepts and the models of resource brokering in Kubernetes, giving a more precise high-level definition of the roles and functions involved in the brokering process. Firstly, we define the concepts of Multi-Tenant and Multi-Cloud. Then we go through some user stories and use cases for resource brokering in cloud services federations, focusing on some real scenarios, proceeding to derive an understanding of possible added-value services, and finally devise Liqo-based solutions to offer these services, dwelling in particular on the "Catalog" model which will be studied deeply in the following.

7.1 MULTI-TENANCY AND MULTI-CLOUD

To provide a common ground for our studies, we report here the definition of Multi-Tenancy and Multi-Cloud.

Multi-tenant architecture

A Multi-tenant architecture represents a single software instance that serves multiple and heterogeneous customers. Each customer is called a tenant, who has the possibility to act in a restricted functional domain.

This type of architecture is able to work because each tenant is physically integrated together with all the others, but is logically separated from them. This concept has been adopted above all in Cloud adoption and is used most with cloud computing, in particular in order to allow each tenant's data to be separated from each other. This has therefore given the possibility, for example, to use the same server to host multiple users, and ideally provide them a secure space to store data.

Of course, this approach has brought several advantages in terms of costs and scalability, but it has also introduced some disadvantages. In particular, it is clear to imagine how the general complexity of development has grown considerably, especially from a security and resource management point of view.

MultiCloud

Funded by Horizon Europe Framework Programme of the European Union

The concept of Multi-Cloud, sometimes also used in the sense of Hybrid Cloud, is the common background. It simply means a company, or more generally an enterprise, which during the deployment process makes use of various heterogeneous cloud providers, including On-Premise, instead of applying a one vendor business approach. This allows the company not only to be able to differentiate the services offered and therefore improve their reliability and redundancy, but also reduces vendor lock-in, thus impacting the competitiveness of the markets and potentially also costs. If we imagine all this in a contemporary context, in which the control and sovereignty of data plays a fundamental role both at a geopolitical level and at a strategic level in the competition between companies, it makes it easy to understand how this principle is becoming more and more fundamental in the approach to the modern Cloud.

7.2 USER STORIES

The need for resource brokers arises from creating large-scale Kubernetes clusters for the purpose of sharing computing resources. Brokers address some shortcomings of the peer-to-peer, horizontal model in scaling to large numbers of users: as the number of providers increases, it becomes increasingly harder to have full visibility over the resources and organizations present on the network. Vice versa, each provider may no longer place the same level of trust in consumers as in a smaller network with well-known participants. Furthermore, even if each party has full visibility over the global resources, an "overseeing organization" may be able to optimize the distribution of workloads using proprietary metrics and the knowledge of each customer's needs.

From this consideration we can contextualize the broker as a commercial operator in the niche of enabling multi-cloud environments, that provides added-value services to large, pre-existing federations. As such, the broker can additionally be understood in the GAIA-X framework as a participant that fulfills Federation Services: we will see that our work addresses existing GAIA-X Federation Services like Access Management, but also expands the notion of Federated Catalogues to include IaaS offerings.

In this respect, TOP-IX (Torino Piemonte Internet eXchange), the Internet Exchange Point for north-western Italy, is seeking to expand its business in the direction of being a neutral, trusted intermediary to data exchange and cloud computing, extending its role consistently with the historical nature of IXPs.

One of the most relevant projects that we worked on is Structura-X[6]: this project was born as the lighthouse project of Gaia-X, the European cloud initiative. The goal of Structura-X is to create a federated infrastructure that will include both cloud providers and IXPs, enabling a trustful environment for the exchange of data and services. The idea is to create an ecosystem

[6] https://www.top-ix.org/en/2021/12/20/structura-x-lighthouse-project-for-european-cloud-infrastructure/.

Funded by Horizon Europe Framework Programme of the European Union

capable of competing with the hyperscalers, both in terms of networking and in terms of computing layer. The first demo of the project was presented in Paris in November during the Gaia-X Summit 2022. On this occasion, we presented the first version of the Structura-X architecture, and also of the project studied, developed and implemented in this work.

7.3 GENERIC SCENARIOS

Our research shows that there are several different scenarios that address different needs. We distinguish between B2C and B2B and between the Provider and Customer (respectively who provides the resources/services and who uses them).

7.3.1 B2C

7.3.1.1 Scenario 1 – Discovery

There are a multitude of cloud operators, each offering different features and having their own clusters. The "cloud market" is a sprawl with little discoverability integrated into the network.

- *Customer story*: I want to discover what providers are available, so that I can choose the one that best fits my needs.
- *Provider story*: I want to advertise my resources to potential customers in a way that is integrated with the federation.

7.3.1.2 Scenario 2 - Metrics and certification

There are many more cloud operators, to the point that it is infeasible or undesirable for a customer to choose the optimal one. Furthermore, some metrics of interest (e.g., latency) may not be available to the user, or may be self-certified.

- *Customer story*: I want to choose the optimal cluster, but I do not have the time/data to do it myself.
- *Broker story*: I want to certify metrics like latency and uptime, so that I provide a value-added service.
- *Provider story*: I want to advertise my resources, and compete with other providers with certified, reliable metrics.

7.3.1.3 Scenario 3 - Data sovereignty

There are several organizations that want to enable access and computation over their data while retaining control over it.

Funded by Horizon Europe Framework Programme of the European Union

- *Customer story*: I want to run computations over sensitive data (e.g., healthcare databases).
- *Provider story*: I want to make my data available, but I also want to make sure it is in safe hands and in compliance with regulations.
- *Broker story*: I want to certify customers and workloads, so that I provide a value-added service.

7.3.2 B2B

7.3.2.1 Scenario 1 – Competitiveness

There are several small cloud providers that are individually not competitive with larger players.

- *Customer story*: I want to access a large amount of computing resources.
- *Provider 1 story*: I want to join forces with other providers to enable a larger, aggregated commercial offering.
- *Provider 2 story*: I want to increase my visibility in the cloud market to compete with bigger players and engage new customers.

7.3.2.2 Scenario 2 - Service offering extension

There are several cloud providers that want to extend their service offering on demand without investing to create a suitable proprietary infrastructure and/or without losing potential customers.

- *Customer story*: I need a set of specific services not entirely offered by my cloud provider.
- *Provider 1 story*: I want to offer to my customers certain services through my infrastructure by engaging them on demand from other vendors.
- *Provider 2 story*: I want to increase my business by providing directly to other vendors some service offers, exploiting infrastructural resources that would be wasted.

7.3.2.3 Scenario 3 - GDPR Data compliance

There is data that, according to GDPR, needs to be stored in a specific geographical location compliant with the rule, or that, for some reasons, cannot be taken outside some political or geographical boundaries.

- *Customer story*: I want to store my data in a place compliant with certain rules and simultaneously access it from other services hosted in a separate infrastructure of the same cloud provider.

Funded by Horizon Europe Framework Programme of the European Union

- *Provider 1 story*: I want to offer to my customers a data space service that respects their needs joining an IaaS offer provided by another vendor, thus guaranteeing them a continuity of service.
- *Provider 2 story*: I want to increase my business providing to other vendors some specific IaaS GDPR compliant offers.

7.4 Real scenarios

7.4.1 Gaia-X

Gaia-X is a project reportedly working on the development of a federation of data infrastructure and service providers for Europe to ensure European digital sovereignty. It seeks to create a proposal for the next generation of data infrastructure for Europe, as well as foster the digital sovereignty of European cloud service users. It is reportedly based on European values of transparency, openness, data protection, and security. To accomplish this, it hopes to specify common requirements for a European data infrastructure and develop a reference implementation.

7.4.2 Structura-X

A lighthouse project for European cloud infrastructure endeavors to enable existing Cloud Service and Infrastructure Providers (CSP) data and infrastructure services to be Gaia-X certifiable. The goal is to create an ecosystem of independent CSPs, orchestrated by a shared layer of federation certification and labeling services based on Distributed Ledger Technology (DLT). The first milestone on the Structura-X project is to provide and certify Infrastructure Service Offerings against the definitions of the Gaia-X Trust Framework to provide Transparency and Trust to all participants.

7.5 Model and use-cases

After analyzing the possible scenarios, it is now clear how the existence of a third entity, the broker, is important and necessary. It acts as a centralized binder between the various subjects taken into consideration. From the user stories we can therefore develop, based on the role and the functionalities they will cover, at least three possible brokering solutions, which are mainly distinguished by the wideness of control they have in the management of the relationships between Providers and/or Customers.

These correspond to three different typologies and roles that are mostly independent with one another, which need to be developed into three software components:

Funded by Horizon Europe Framework Programme of the European Union

- **The Catalog**: this is represented by an endpoint that customers and providers can browse and query. The main role is that of advertisement and discovery, both from the point of view of the individual subjects existing in the federation and of the services offered and made available within the same. The goal is therefore to rely on broker to be informed of what clusters are on the network, what are their features and resources, and possibly other information (e.g., a trusted estimate of their uptime or latency). Thanks to this broker all the subjects can join available offers and establish new interconnections on demand. Each interconnection is completely independent from the broker, it is based on a peer-to-peer model, thus keeping the overall ecosystem completely decentralized by eliminating the possibility that the broker may somehow be a single point of failure.
- **The Orchestrator**: an active component that is in charge only of scheduling computing resources according to defined policies. This implies a well-defined splitting between control planes, up to the orchestrator, with data planes, demanded to the involved customers. So, users rely on brokers to orchestrate their workloads, either because they do not want to deal with the complexity of orchestration (for an extreme example, a customer may not even use Kubernetes, instead deploying Helm charts from the broker) or because the broker has access to proprietary information that can provide for an optimal orchestration. This process can also go the other way: providers can require customers to go through a trusted orchestrator that acts as a security gateway to inspect the users' identities or their workloads.
- **The Aggregator**: Unlike the previous two, this broker is completely opaque, acting as a single virtual cluster, in charge of totally managing control and data plane of the clusters below him. Providers rely on brokers to present their resources and those of their partners in aggregated form, creating a commercial offering that can compete with larger and more established providers. We can identify this broker as a middleman that presents a unified view of resources.

7.6 Catalog

Now that we have a clearer picture of our design goals, let us understand how the standard Kubernetes resources and the Liqo paradigm can be used to accomplish these tasks, and how they were extended. We note that much of the functionality in the "peer-to-peer version" of Liqo can be reused to a large extent, at least for a proof of concept. The only parts that require substantial changes are the resource enforcement for the management of multi-tenant peerings and the customization of the peering resource reservation in such a way as to bypass the standard process in which Liqo calculates and allocates resources.

At the functional level, a catalog needs to receive peering credentials and the commercial offers of each provider cluster, store and aggregate them, and send the full list to each one

Funded by Horizon Europe Framework Programme of the European Union

that wants to browse the catalog when a query is issued. The aspect of peering credentials is arguably simpler.

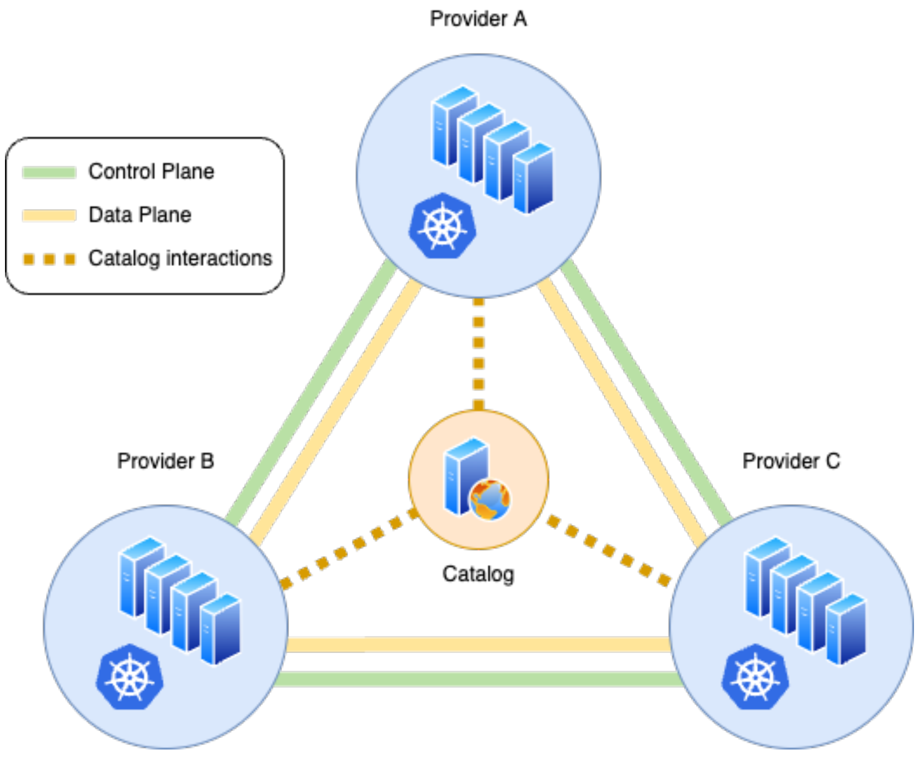


Figure 26. Catalog example.

Peering takes place over IP with an optional authentication step based on a token. As such, it is sufficient for a cluster to know the provider's IP address, its cluster ID, and its authentication token. With this information the provider/customer can run Liqo peering command to establish the connection, and Liqo will proceed to do the rest. The representation of cluster resources is more complex depending on the specific business requirements: Liqo supports sharing hardware resources with a format that encapsulates Kubernetes' ResourceQuotas, but a customer may also be interested in SaaS offerings or in certified metrics for uptime/latency/etc.

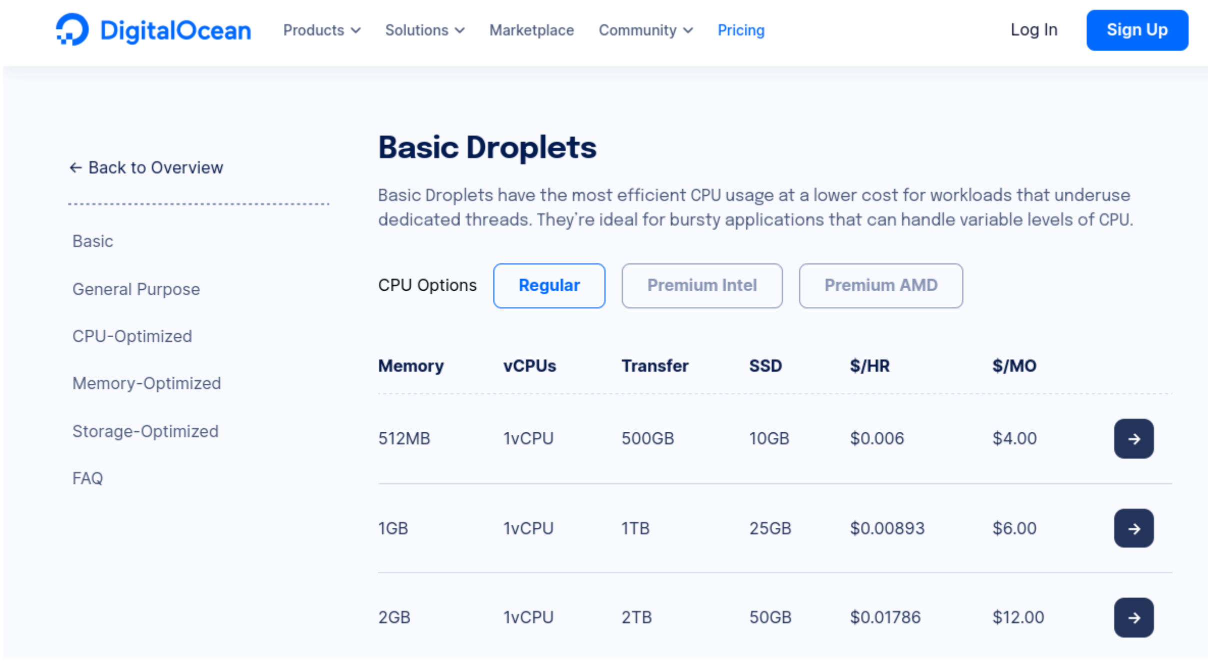
In our work we will concentrate only on the first case, trying to develop an IaaS environment.

We might be tempted to reuse parts of the Liqo peering logic in our catalog. In a one-to-one peering, sharing the list of hardware resources is a substantial part of creating the peering: after authentication is carried out and networking is set up, the consumer creates a ResourceRequest in the provider cluster, which in turn creates a ResourceOffer in the customer cluster to signal acceptance, for a better understanding we demand the reader to the Liqo Official documentation. (This architecture is intended to support a resource negotiation process, as hinted by the CR naming, but at the time of writing it is not possible to encode queries in the ResourceRequest or to limit the resources offered.) A catalog would then need to collect ResourceOffers from all providers, but in doing so it will need to open

Funded by Horizon Europe Framework Programme of the European Union

one peering for each provider. This makes for a rather unwieldy solution in terms of resource usage, not to mention that the broker needs to run a Liqo instance (and possibly a Kubernetes stack) exclusively for collecting ResourceOffers. We propose an alternative, light-weight solution where we decouple the resource advertisement from the peering process. Indeed, if we define a stand-alone protocol for advertisements and queries, the catalog only needs to support our protocol. Additionally, decoupling these two processes enables a range of complex resource negotiation logics, ranging from unconditional acceptance (which might be desired in simple, one-to-one peerings where negotiations happen on paper) to extensible and dynamic marketplaces (which we envisage in a wide computing federation).

Our protocol conceptualizes resource offers as a multitude of "packages," each one with a set of hardware resources as well as, potentially, SaaS and other immaterial resources. We call each "package" a plan, and a collection of plans is an offer. This reflects the offer structure of commercial computing providers like Amazon AWS, Azure or DigitalOcean: there are several offers optimized for different workloads, and each offer has a number of plans that determine the size. We will go in depth with the description of this model in the following, where we also explain the evolution of this model in a distributed fashion and of course we describe how all the components interact with each other and elaborates and/or exchange data.



Basic Droplets

Basic Droplets have the most efficient CPU usage at a lower cost for workloads that underuse dedicated threads. They're ideal for bursty applications that can handle variable levels of CPU.

Memory	vCPUs	Transfer	SSD	$/HR	$/MO
512MB	1vCPU	500GB	10GB	$0.006	$4.00
1GB	1vCPU	1TB	25GB	$0.00893	$6.00
2GB	1vCPU	2TB	50GB	$0.01786	$12.00

Figure 27. Excerpt from the Digital Ocean catalog.

7.7 ORCHESTRATOR

Recall the user story described at the beginning of this chapter (B2C - Scenario 2). To the consumer, orchestration is a value-added service by which optimal providers are chosen

Funded by Horizon Europe Framework Programme of the European Union

according to some (possibly private) metrics; to the provider, it is a tool for competing with certified metrics, as well as - potentially - a security filter in front of consumers.

We can define an Orchestrator as an intermediary that operates on the control plane, with the primary function of enforcing some policy in the peering process and in the distribution of workloads. This policy may take several forms, ranging from optimizing metrics to authentication and authorization constraints, but a common feature is that the Orchestrator takes the additional responsibility in actuating a policy; compare this with the Catalog, which is merely a passive component that provides non-binding suggestions.

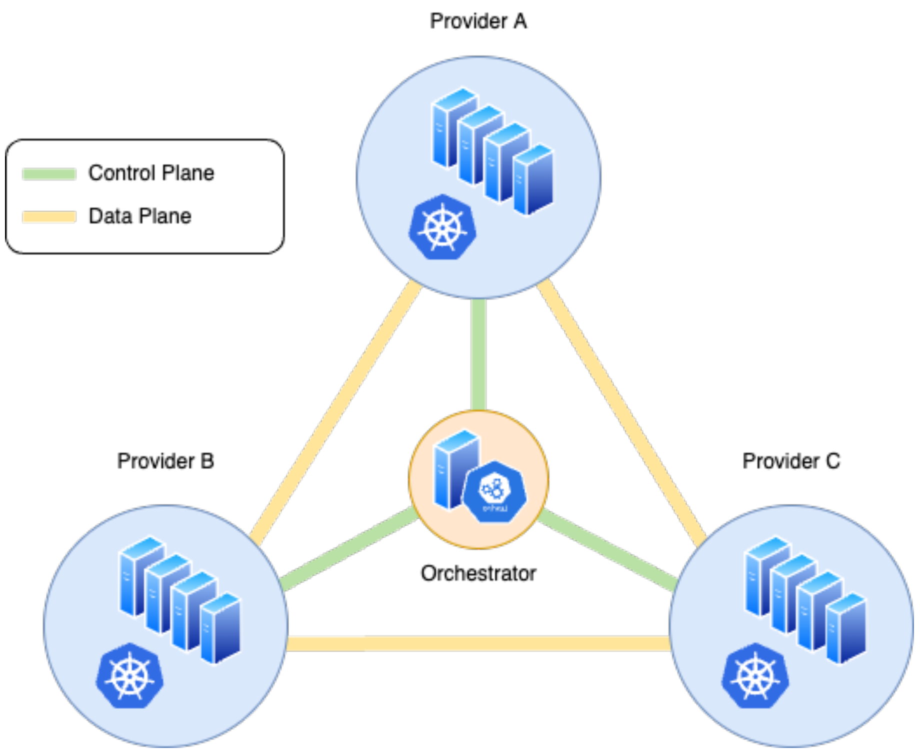


Figure 28. Orchestrator example.

At the implementation level, the Orchestrator is able to enforce arbitrary policies if we establish that all peerings must go through it. At the same time, it is not entirely "opaque": we want customers to retain visibility into what providers their pods are being offloaded to, and vice versa, we want a provider to know what customers it is serving. In fact, performance-wise it would be ideal if the data plane was direct between the customer and the provider, while the control plane retains the Orchestrator as an intermediary: if on the other hand the data plane had to pass through the broker, the latency would suffer from the extra hop, and the broker would have to allocate sufficient bandwidth for serving all its customers. In essence, if we want to develop a scalable brokering solution it is a prerequisite that we decouple the control plane from the data plane, so that only the control plane may be proxied.

Funded by Horizon Europe Framework Programme of the European Union

7.8 AGGREGATOR

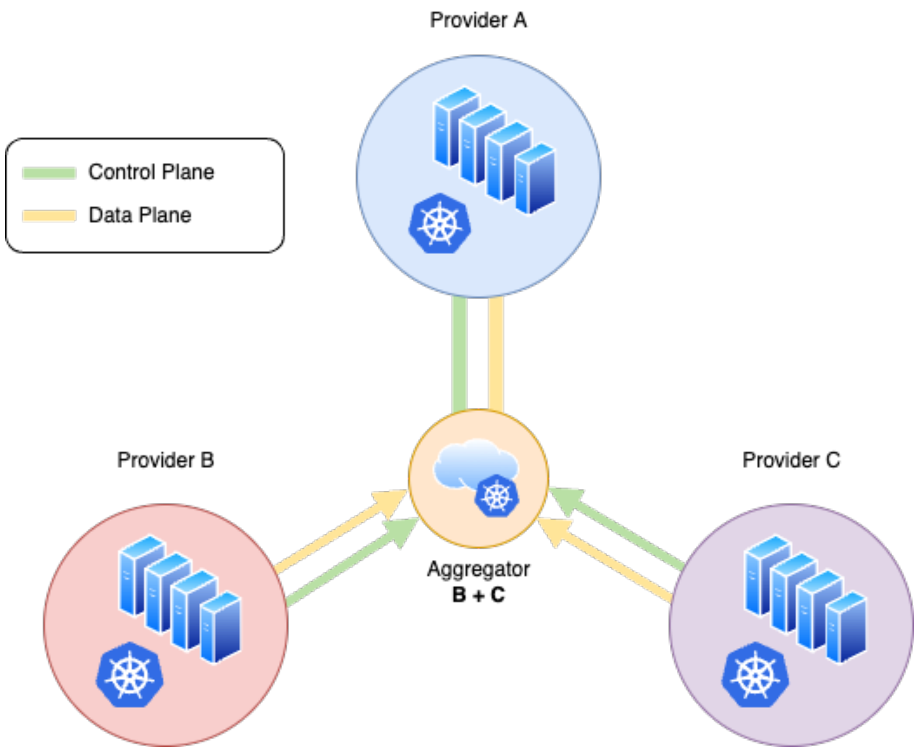


Figure 29. Aggregator example.

At the beginning of this chapter we explored some user stories that can be used to motivate the need for a brokering solution. This section will focus on the use case (B2B - Scenario 1) of a cloud provider that wants to be competitive in the market by joining forces with other cloud providers (in the Figure Provider B and C), to be able to offer a better service to its customers (Provider A) or just to increase its market share and visibility.

In this scenario, the brokering model that best suits the needs is the Aggregator model, because it allows the cloud provider to federate its resources with other cloud providers, and to offer them in the market as a single entity. In this way, the cloud provider can offer a better service to its customers, maybe with a better price, performance, or with a better service diversification.

Unlike the Orchestrator, in this case both the "Control plane" and the "Data plane" of the cloud providers are proxied and controlled by the Aggregator, which is in charge of aggregating their resources, and of distributing the workloads to the ones that are able to handle them based on some metrics and policies. On top of that it exposes to the customers, and so to the market in an opaque way, the aggregated resources of the providers that are federated with it. The final result is that no one knows and matters about the exposition of its services and resources to the market, and how the workloads are distributed among the providers federated with the Aggregator. They just should define their own policies, if they

Funded by Horizon Europe Framework Programme of the European Union

want, or let the Aggregator decide where based on metrics. This model implies of course a high responsibility for the Aggregator, because it becomes the single point of management of the resources of the providers that are federated with it. Moreover, it has to be sure that the resources of the providers are not overused, and that the workloads are distributed in a fair way among the parties. To do that, the Aggregator has to be able to monitor the resources and metrics of the overall environment.

From a security and reliability point of view, it should appear as a single point of failure, in fact if the Aggregator fails, all the providers that are federated with it will not be reachable anymore. To avoid this, a complex system of redundancy and failover should be implemented, and the Aggregator should be able to recover from failures in a fast way. Moreover, the providers should be able to bypass the Aggregator in case of failure, or provide backup peerings with other instances of the Aggregator.

7.9 MODEL COMPARISON

Table 3. Comparing different brokering models.

Model	Peering control plane	Peering Data Plane	Workloads scheduling	Resource negotiation	Nodes knowledge	Nodes aggregatio n
Catalog	No	No	No	Maybe	Only Public	No
Orchestrator	Yes	No	Yes	Maybe	Public and owned	No
Aggregator	Yes	Yes	Yes	Maybe	Public and owned	Yes

Legend

- **Control Plane**: capability to control and manage Peering control plane.
- **Data Plane**: capability to control and manage Peering data plane.
- **Workloads scheduling**: capability to manage workloads scheduling between nodes.
- **Resource negotiation**: depending on the purpose, the capability to negotiate resources and instantiate peerings.
- **Nodes knowledge**: depending on the purpose, capability to know the overall ecosystem architecture (intra domain or inter domain).
- **Nodes aggregation**: capability to aggregate node resources and obfuscate them as a unique entity/node.

Funded by Horizon Europe Framework Programme of the European Union

8 FLUIDOS AT THE EDGE

This Section proposes a possible architecture for FLUIDOS when running at the edge nodes, whose characteristics are very different from what we can find on servers, and even on common desktop/laptop computers.

8.1 IoT Edge Granularity

In the context of the Internet of Things (IoT) different layers have been defined to capture the hierarchy of processing and decision-making capabilities that extend from the edge of the network (i.e., the devices) to the cloud, as shown in the picture below. Overall, the main difference between these layers is their location and capabilities of the computing resources involved, with micro edge and deep edge architectures involving smaller, lower-power devices located closer to the data source, and meta edge and fog edge architectures involving larger, more powerful devices located further from the data source.

In fact, IoT environments are inherently consisting of different devices such as microcontroller units (MCUs) or multiprocessor units (MPUs) from several manufacturers with different architectural features (32-bit/64-bit), some with integrated accelerators, graphics processing units (GPUs), tensor processing units (TPUs), machine learning (ML) cores, cryptographic cores etc.), and integrated sensors. From the software standpoint, they have different operating systems, ranging from bare metal applications to real-time OS, and general-purpose OS (Linux, Android, etc.), thus forming a heterogeneous resource fabric.

Although we can observe a new trend in which the above edge devices are becoming smarter and smarter, thus possibly enabling to (partially) move the intelligence from the cloud toward the edge, still this heterogeneity does not allow a one-fits-all approach. Therefore, next sections will introduce the main characteristics of each layer, which provides the ground for defining our solution in FLUIDOS.

Funded by Horizon Europe Framework Programme of the European Union

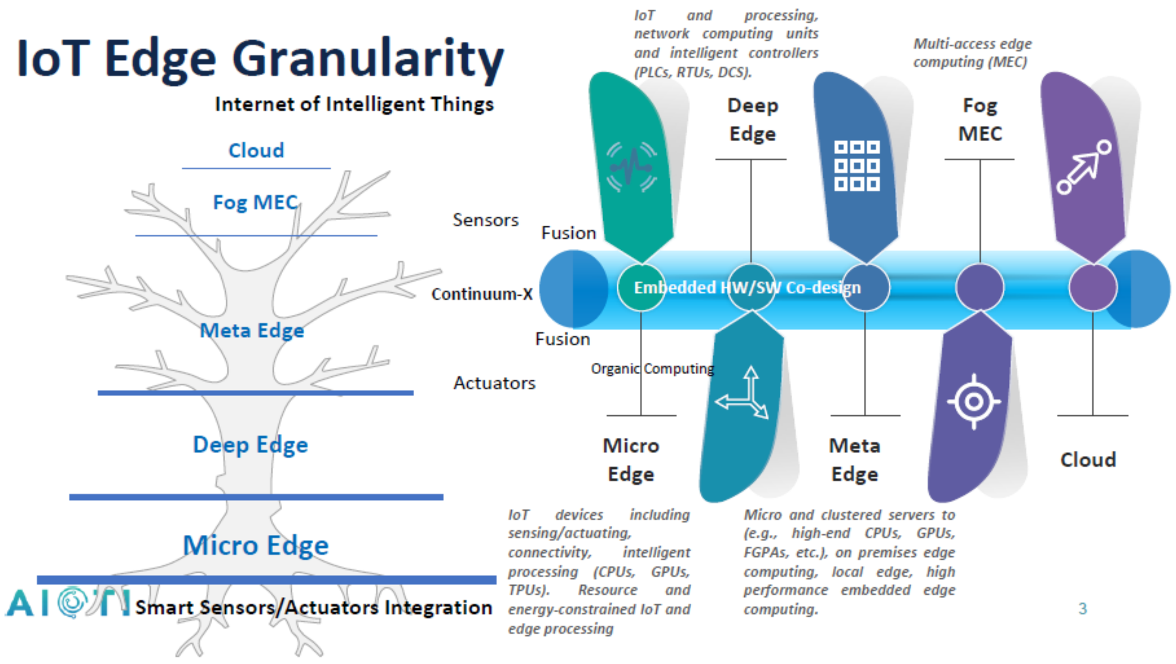

IoT Edge Granularity
Internet of Intelligent Things
Cloud
Fog MEC
Meta Edge
Deep Edge
Micro Edge
AIOTI Smart Sensors/Actuators Integration
Sensors
Fusion
Continuum-X
Fusion
Actuators
Organic Computing
Embedded HW/SW Co-design
IoT and processing, network computing units and intelligent controllers (PLCs, RTUs, DCS).
Deep Edge
Multi-access edge computing (MEC)
Fog MEC
Micro Edge
Meta Edge
Cloud
IoT devices including sensing/actuating, connectivity, intelligent processing (CPUs, GPUs, TPUs). Resource and energy-constrained IoT and edge processing
Micro and clustered servers to (e.g., high-end CPUs, GPUs, FGPAs, etc.), on premises edge computing, local edge, high performance embedded edge computing.
3

Figure 30. IoT Edge granularity.

8.1.1 Micro Edge

The micro edge refers to the lowest level of the hierarchy, which includes the devices and sensors that are located at the edge of the network. These devices typically have limited processing and storage capabilities, and they rely on the other layers of the hierarchy for communication and decision-making. These services include sensing/actuating, connectivity, intelligent processing (CPUs, simple GPUs, TPUs).

Devices belonging to the micro edge generally lack proper virtualization support, however these devices can be dynamically attached to different edge devices or gateways. This is usually possible due to the networking infrastructure. MCUs with integrated sensors are commonly connected to a bus (e.g., Modbus, CAN-bus) or via wireless protocols (e.g., Bluetooth, LoRaWAN, WIFI), which allow different entities to collect data from a swarm of such devices. Logical grouping can facilitate energy-efficient, reliability, quality-of-results, or performance goals when collecting sensor data. For example, collecting data (temperature, humidity) from sparse LoRaWAN-based sensors in a large field in agriculture gives a fast overview, but of low confidence or detailed view. Moreover, the use of many cameras or proximity sensors in a vehicle to gather data involves trade-offs between real-time reaction, decision-making accuracy, and energy usage. Further, new sensors can be arbitrarily added to existing equipment to produce new data streams, leading to stream-based analysis operations for which there is not currently any historical data.

Funded by Horizon Europe Framework Programme of the European Union

8.1.2 Deep Edge

The next layer, called Deep Edge, refers to the intermediate layer of the hierarchy, which is located between the micro edge and the meta edge. This layer is responsible for more advanced processing and decision-making, and it may include devices such as gateways and edge servers that are capable of handling more complex tasks than the devices at the micro edge. It includes IoT devices and processing, network computing units and intelligent controllers (PLCs, RTUs, DCS, IoT Gateways).

The IoT Gateways is a device that connects micro edge devices to the IP network, allowing them to communicate with each other and with other devices and systems over the Internet or to the cloud. It serves as a bridge between the local non-IP network to the IP network, providing a secure and reliable connection for the micro edge devices to access the Internet and to send and receive data. Also, it expands spatial connectivity capabilities of the already IP aware micro edge devices (Matter, Thread etc.) The gateway typically includes hardware and software components that enable it to collect data from the local devices, process and filter the data, and transmit it to the Internet or other destination. It may also include features such as device management, security, and analytics. IoT gateways are used in a variety of applications, including industrial automation, smart homes, and smart cities. For instance, in the smart manufacturing these devices are programmable logic controllers (PLC) or distributed control systems (DCS) connected to sensor and actuators (e.g., robot, environmental control etc.) via a field bus or control system while providing an IP connectivity

8.1.3 Meta Edge

The next layer, called Meta Edge, is used to describe edge computing architectures that involve a layer of computing resources located between the edge and the cloud. The meta edge may be responsible for aggregating and processing data from multiple deep edge devices, and it may be located closer to the edge than the cloud, but further from the data source than micro edge or deep edge devices. This layer includes Micro and clustered servers to (e.g., high-end CPUs, GPUs, FGPAs, etc.), on premises edge computing, local edge, high performance embedded edge computing.

In the scope of FLUIDOS, the Meta-Edge layer is responsible to enable connecting and controlling the edge (or leaf) devices and thus offer decoupling of application development from device data access, provision of complete lifecycle data management for devices.

8.1.4 Fog/MEC and Cloud Computing

Next layers further moves toward distant resources, with the components called Fog Computing (usually at the customer's premises) and Multi-access Edge Computing (MEC) (typically at the telco side), till to the Cloud, which brings large data centers into the picture.

Funded by Horizon Europe Framework Programme of the European Union

The Fog/MEC layer includes a layer of computing resources located between the Meta edge and the cloud, similar to the meta edge. However, devices in the fog edge are typically used to describe architectures that involve a larger number of more powerful computing resources (e.g., in the order of 10+ servers), and it is often used in applications that require real-time processing of large amounts of data.

With emerging technologies like edge and fog computing that allow the execution of machine learning models on tiny and lightweight devices, the localization of data processing activities can shift closer to the real sources. As a result, the number of heterogeneous devices in the edge cloud continuum is increasing quickly in terms of the hardware components they include (e.g., multiple CPU architectures, CPU cores, RAM, or accessible accelerators like GPUs). Hence, collecting and transmitting sensor data, as well as pre-processing and actuation, are all included in the definition of a service, a resource that is potentially assigned to a containerized cloud application. Exposing such data streams to Kubernetes clusters or scaling the device data across multiple nodes can provide seamless management of edge and micro-edge devices.

8.2 Use-case Scenarios for FLUIDOS at the Edge

TODO

8.3 Technical Architecture

One approach to create a continuum from the cloud to the edge can be done by using KubeEdge (https://kubeedge.io/en/) enabling the deployment of containerized applications on edge devices, and to enable these applications to communicate with other devices and systems in the network, including cloud-based systems. The KubeEdge architecture consists of several components that work together to enable the deployment and management of containerized applications on edge devices, as described in Section 5.1.1.

On the basis of the KubeEdge architecture, the device controller is the cloud component of KubeEdge which is responsible for device management. By using the Kubernetes Custom Resource Definitions (CRDs), it describes device metadata/status and synchronizes these device updates between edge and cloud. The edge controller is the bridge between Kubernetes API server and EdgeCore, while CloudHub links controllers the edge part over and supports both web-socket based connection as well as a QUIC[7] protocol access at the same time.

[7] Iyengar, J., Ed., and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", 2021.

Funded by Horizon Europe Framework Programme of the European Union

The Mapper is a part of the agents (edge side), interfacing different leaf devices connection protocols, that creates a physical connection between edge and an IoT device by acting as an asynchronous message broker using MQTT. The Mapper device protocol plugins automatically convert from the standard message to the real message that the device understands. The messages provided to Mapper are picked up by a component on the edge node called Edge Core, which then transmits them to a container that is active on the edge or the cloud side. It is also responsible for a variety of other tasks, such as checking device health and collecting telemetry data.

The Device Twin, a cloud-based digital representation of a physical device, enables local autonomy by maintaining meta-data for each device (when data/control, e.g., device attributes and status, are delivered by the cloud to the edge for each specific device), so that the EdgeCore can recover them when a device or the edge is disconnected, or when the edge node restarts. The Meta Manager enables communication with containerized applications running in pods and is responsible for storing/retrieving metadata related to them.

While huge number of connected devices continuously capture and send data for analysis and decision making in the cloud, and well-endorsed projects, e.g., K3s provide fully functional Kubernetes instances to edge devices, efficient data management schemes at the edge, increasing degree of heterogeneous IoT devices, coexistence of wireless technologies and protocols, specific operating systems and services make it a challenging task.

Funded by Horizon Europe Framework Programme of the European Union

9 CONCLUSIONS

This Deliverable presents the initial work carried out in the first phase of WP2, namely FLUIDOS scenarios, requirements, and the resulting reference architecture, in addition to a detailed presentation about the unique vision of the computing continuum, also called "liquid computing", proposed in FLUIDOS.

This document highlights also the considerable amount of work has been dedicated to the state-of-the-art, and in particular what already exists, and which are the components and features that are missing to bring to life the vision of the FLUIDOS project.

This document will be updated and further extended in the next months, to incorporate (1) new findings (e.g., refined requirements, new scenarios, architectural updates), (2) possible feedback from early adopters, and (3) potential research collaborations with other projects funded in the same MetaOS call.

Funded by Horizon Europe
Framework Programme of the European Union

10 REFERENCES

[1] C. Pahl, “Containerization and the PaaS cloud”, Jul. 2015.
[2] CNCF Staff, “CNCF annual survey 2021”, 2021
[3] P. Garcia Lopez et al., “Edge-centric computing: Vision and challenges”, 2015
[4] W. Shi et al., “Edge computing: Vision and challenges”, 2016
[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things”, 2012
[6] D. Milojicic, “The edge-to-cloud continuum”, 2020
[7] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, “A unified model for the mobile-edge-cloud continuum”, 2019
[8] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, “Instability in geo-distributed kubernetes federation: Causes and mitigation”, 2020
[9] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, “Impact of etcd deployment on kubernetes, istio, and application performance”, 2020
[10] L. Osmani, T. Kauppinen, M. Komu, and S. Tarkoma, “Multi-cloud connectivity for kubernetes in 5g networks”, 2021
[11] L. Leong, “Comparing cloud workload placement strategies”, 2020
[12] D2IQ, “Multi-cluster management: Reduce overhead and redundant efforts”, 2021
[13] P. Mell and T. Grance, “The NIST definition of cloud computing”, 2021
[14] A. Yousafzai et al., “Cloud resource allocation schemes: Review, taxonomy, and opportunities”, 2017
[15] R. Buyya et al., “A manifesto for future generation cloud computing: Research directions for the next decade”, 2018
[16] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. D. Turck, “Resource management in a containerized cloud: Status and challenges”
[17] R. Bias, “Architectures for open and scalable clouds”, 2022
[18] CNCF Staff, “CNCF survey 2020”, 2020
[19] https://k3s.io
[20] K. Goldenring, “Announcing akri, an open source project for building a connected edge with kubernetes”, 2020
[21] Z. Zheng et al., “An overview on smart contracts: Challenges, advances and platforms”, 2020
[22] J. Fritsch and C. Walker, “The problem with data”
[23] https://kubeedge.io/en/
[24] https://karmada.io/
[25] https://liqo.io/
[26] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. Pervasive Computing, IEEE, 8(4):14 –23, 2009

Funded by Horizon Europe Framework Programme of the European Union

[27] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," IEEE Cloud Computing, vol. 3, no. 6, pp. 76–83, nov 2016.
[28] https://github.com/kubernetes-sigs/kubefed
[29] https://admiralty.io/
[30] https://www.lfedge.org/projects/eve/
[31] https://threefold.io/
[32] https://www.balena.io/os
[33] https://github.com/aurae-runtime/aurae
[34] https://submariner.io/
[35] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in Stabilization, Safety, and Security of Distributed Systems, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.
[36] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, ser. PODC 87. New York, NY, USA: Association for Computing Machinery, 1987, p. 112. [Online]. Available: https://doi.org/10.1145/41840.41841
[37] https://repository.tudelft.nl/islandora/object/uuid%3A5fe8d907-a98c-4364-b594-69ebb044767e
[38] Ometov A, Molua OL, Komarov M, Nurmi J. A Survey of Security in Cloud, Edge, and Fog Computing. Sensors. 2022; 22(3):927. https://doi.org/10.3390/s22030927
[39] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, Torsten Eymann, Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust,Computers & Security,Volume 110,2021,102436,ISSN 0167-4048, https://doi.org/10.1016/j.cose.2021.102436.
[40] Christopher Allen. 2016. The Path to Self-Sovereign Identity. http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html
[41] European Self Sovereign Identity framework (eSSIF) available at https://www.eesc.europa.eu/sites/default/files/files/1._panel_-_daniel_du_seuil.pdf
[42] World Wide Web Consortium (W3C), "Decentralized Identifiers (DIDs) v1.0: Core Data Model and Syntaxes," 2019. Available at https://w3c.github.io/did-core/#did-document
[43] World Wide Web Consortium (W3C), "Verifiable Credentials Data Model v1.1" Available at https://www.w3.org/TR/vc-data-model
[44] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. 2018. A survey on essential components of a self-sovereign identity. , 80–86 pages. https://doi.org/10.1016/j.cosrev.2018.10.002
[45] Yu L, He M, Liang H, Xiong L, Liu Y. A Blockchain-Based Authentication and Authorization Scheme for Distributed Mobile Cloud Computing Services. Sensors. 2023; 23(3):1264. https://doi.org/10.3390/s23031264
[46] Nikos Fotiou, Iakovos Pittaras, Vasilios A. Siris, George C. Polyzos. 2019. Enabling Opportunistic Users in Multi-Tenant IoT Systems using Decentralized Identifiers and

Funded by Horizon Europe Framework Programme of the European Union

Permissioned Blockchains. In 2nd Workshop on the Internet of Things Security and Privacy (IoT S&P'19), November 15, 2019, London, United Kingdom. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3338507.3358622

[47] Peterson, Gunnar. "Don't Trust. And Verify: A Security Architecture Stack for the Cloud." IEEE Security & Privacy 5 (2010): 83-86.

[48] S. Kahvazadeh, X. Masip-Bruin, R. Diaz, E. Marín-Tordera, A. Jurnet and J. Garcia, "Towards An Efficient Key Management and Authentication Strategy for Combined Fog-to-Cloud Continuum Systems," 2018 3rd Cloudification of the Internet of Things (CIoT), Paris, France, 2018, pp. 1-7, doi: 10.1109/CIOT.2018.8627111.

[49] https://aws.amazon.com/kms/

[50] https://azure.microsoft.com/en-us/services/key-vault/

[51] https://cloud.google.com/kms/

[52] AMD White paper, "AMD SEV-SNP: Strengthening VM Isolation withIntegrity Protection and More", January 2020

[53] Hibbert, M., Hu, F., & Hu, W. (2016, November). Intel SGX explained. In Proceedings of the 2016 ACM Asia Conference on Computer and Communications Security (pp. 196-207). ACM

[54] https://aws.amazon.com/nitro-enclaves/.

[55] https://cloud.google.com/confidential-computing.

[56] Jinyu Gu, et.al., Secure Live Migration of SGX Enclaves on Untrusted Cloud. DSN 2017

[57] Fritz Alder, et.al., Migrating SGX Enclaves with Persistent State. DSN 2018

[58] Confidential container documentation: https://github.com/confidential-containers/documentation

[59] https://www.fortanix.com/solutions/use-case/confidential-computing

[60] Red Hat. (2022). OpenShift Container Platform. Retrieved from https://www.redhat.com/en/technologies/cloud-computing/openshift

[61] Gao et al. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. (DSN 2017)

[62] Gao et al. A Study on the Security Implications of Information Leakages in Container Clouds. (IEEE Transactions on Dependable and Secure Computing 2018).

[63] Lin et al. A Measurement Study on Linux Container Security: Attacks and Countermeasures. (ACSAC 2018)

[64] Combe et al. To Docker or Not to Docker: A Security Perspective. (Cloud Computing 2016)

[65] Nam et al. BASTION: A security enforcement network stack for container networks. (USENIX ATC 2020)

[66] Gao et al. Houdini's Escape: Breaking the Resource Reign of Linux Control Groups (CCS 2019)

[67] Khalid et al. Iron: Isolating Network-based CPU in Container Environments (NSDI 2018)

[68] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, 'Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection', in Proceedings of the 2022 on Cloud Computing Security Workshop, 2022.

Funded by Horizon Europe Framework Programme of the European Union

[69] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del-Rincón, and D. Siracusa, 'Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection', IEEE Transactions on Network and Service Management, vol. 17, no. 2, pp. 876–889, Jun. 2020, doi: 10.1109/TNSM.2020.2971776.

[70] Demoulin, H. M., Pedisich, I., Phan, L. T. X., & Loo, B. T. (2018, August). Automated detection and mitigation of application-level asymmetric DoS attacks. In *proceedings of the afternoon workshop on self-driving networks* (pp. 36-42).

[71] Lawal, M. A., Shaikh, R. A., & Hassan, S. R. (2020). An anomaly mitigation framework for iot using fog computing. Electronics, 9(10), 1565.

[72] Zarca, A. M., Bernabe, J. B., Trapero, R., Rivera, D., Villalobos, J., Skarmeta, A., and Gouvas, P. (2019). Security management architecture for NFV/SDN-aware IoT systems. IEEE Internet of Things Journal, 6(5), 8005-8020.

[73] J. Kim et al., "IBCS: Intent-Based Cloud Services for Security Applications," in IEEE Communications Magazine, vol. 58, no. 4, pp. 45-51, April 2020, doi: 10.1109/MCOM.001.1900476.

[74] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza and J. Yusupov, "Automated firewall configuration in virtual networks," in IEEE Transactions on Dependable and Secure Computing, doi: 10.1109/TDSC.2022.3160293.

[75] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2020, pp. 1-7, doi: 10.1109/NOMS47738.2020.9110402

[76] F. Valenza, C. Basile, A. Lioy, D. R. Lopez and A. Pastor Perales, "Adding Support for Automatic Enforcement of Security Policies in NFV Networks," in IEEE/ACM Transactions on Networking, vol. 27, no. 2, pp. 707-720, April 2019, doi: 10.1109/TNET.2019.2895278.

[77] Ferguson-Walter, K., et al.: Game theory for adaptive defensive cyber deception. In: Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security – HotSoS 19, no. April, New York, pp. 1–8. ACM Press (2019)

[78] Al Amin, M. A. R., Shetty, S., Njilla, L., Tosh, D. K., & Kamhoua, C. (2021). Hidden markov model and cyber deception for the prevention of adversarial lateral movement. IEEE Access, 9, 49662–49682

[79] https://metallic.io/threatwise-cyber-deception

[80] Osman, Amr, et al. "Sandnet: towards high quality of deception in container-based microservice architectures." ICC 2019-2019 IEEE International Conference on Communications (ICC). IEEE, 2019.

[81] https://github.com/sustainable-computing-io/kepler

[82] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti and A. Manzalini, "Computing Without Borders: The Way Towards Liquid Computing," in IEEE Transactions on Cloud Computing, doi: 10.1109/TCC.2022.3229163.

[83] James, A., & Schien, D. (2019). A Low Carbon Kubernetes Scheduler. ICT for Sustainability.

[84] https://fleet.rancher.io

Funded by Horizon Europe
Framework Programme of the European Union

[85] https://open-cluster-management.io

[86] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2012. Cloudlets: bringing the cloud to the mobile user. In Proceedings of the third ACM workshop on Mobile cloud computing and services (MCS '12). Association for Computing Machinery, New York, NY, USA, 29–36. https://doi.org/10.1145/2307849.2307858

[87] Goethals, F. De Turck and B. Volckaert, "Extending Kubernetes Clusters to Low-Resource Edge Devices Using Virtual Kubelets," in IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2623-2636, 1 Oct.-Dec. 2022, doi: 10.1109/TCC.2020.3033807.

[88] L. Larsson, H. Gustafsson, C. Klein and E. Elmroth, "Decentralized Kubernetes Federation Control Plane," 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 2020, pp. 354-359, doi: 10.1109/UCC48980.2020.00056.

[89] C. Lauwers, C. Noshpitz, and C. Curescu, “Tosca simple profile in yaml version 1.3.” 2020.

[90] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, “Tosca solves big problems in the cloud and beyond!,” IEEE cloud computing, p. 1, 2018, doi: 10.1109/MCC.2018.111121612.

[91] A. Nejc Bat and L. Korbar, “Xopera: Get your orchestra(tor) pitch perfect.” https://xlab.si/sl/blog/xopera-get-your-orchestrator-pitch-perfect/, 2021.

[92] D. Tamburri, W.-J. Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski, “Tosca-based intent modelling: goal-modelling for infrastructure-as-code,” Sics software-intensive cyber-physical systems, vol. 34, 2019, doi: 10.1007/s00450-019-00404-x.

[93] M. D. Mascarenhas and R. S. Cruz, “Int2it: An intent-based tosca it infrastructure management platform,” in 2022 17th iberian conference on information systems and technologies (cisti), 2022, pp. 1–7. doi: 10.23919/CISTI54924.2022.9820004.

[94] A. S. Jacobs et al., “Hey, lumi! using natural language for Intent-Based network management,” in 2021 usenix annual technical conference (usenix atc 21), Jul. 2021, pp. 625–639. Available: https://www.usenix.org/conference/atc21/presentation/jacobs

[95] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, “Refining network intents for self-driving networks,” in Proceedings of the afternoon workshop on self-driving networks, 2018, pp. 15–21. doi: 10.1145/3229584.3229590.

[96] R. Soulé et al., “Merlin: A language for managing network resources,” Ieee/acm transactions on networking, vol. 26, no. 5, pp. 2188–2201, 2018, doi: 10.1109/TNET.2018.2867239.

[97] M. Riftadi and F. Kuipers, “P4i/o: Intent-based networking with p4,” in 2019 ieee conference on network softwarization (netsoft), 2019, pp. 438–443. doi: 10.1109/NETSOFT.2019.8806662.

[98] A. Angi, A. Sacco, F. Esposito, G. Marchetto, and A. Clemm, “Nlp4: An architecture for intent-driven data plane programmability,” in 2022 ieee 8th international conference on network softwarization (netsoft), 2022, pp. 25–30. doi: 10.1109/NetSoft54395.2022.9844035.

[99] Cilium Cluster Mesh. Available at https://isovalent.com/labs/cilium-cluster-mesh.

Funded by Horizon Europe Framework Programme of the European Union

[100] Open-source VPN for Edge Computing (EdgeVPN.io). Available at https://edgevpn.io/.
[101] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016, "A Validation of DRAM RAPL Power Measurements," Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16). Association for Computing Machinery, New York, NY, USA, 455–470. https://doi.org/10.1145/2989081.2989088.
[102] Karen Scarfone, M.S., Morello, J.: Application Container Security Guide. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf, 2017.

Funded by Horizon Europe Framework Programme of the European Union

Funded by Horizon Europe
Framework Programme of the European Union

11 APPENDIX: HANDLING THE COMPUTING CONTINUUM

This Section reports the huge amount of existing work with respect to the creation of the computing continuum, extending the summary provided in Section 0.

11.1 MOST PROMISING PROJECTS

The idea of providing unified computing resources within a single continuum has been around for a few years, with multiple proposals both from the scientific community and enterprise-driven open-source projects. The common idea is to provide a unified view of multiple distributed clusters either belonging to a single entity or located in a specific geographic area. Scientific papers tackle the problem from different points of view, proposing full-fledged solutions or Proof-of-Concepts (PoCs), either by focusing more on a vanilla approach with respect to the de-facto standard container orchestrator Kubernetes or by implementing their specific set of APIs.

The collection of the literature below represents the state of the art regarding the computing continuum. Production-ready solutions will be prioritized, but a panoramic view on solutions from the research community will be provided as well. A final comparison will be provided, to allow the reader to better compare the respective advantages and disadvantages.

We present firsts the three open-source projects that are most suited as a starting point for the liquid computing idea. We will move then to more experimental projects that, although not the best choice to base FLUIDOS upon, can nevertheless provide nice ideas that can be possibly integrated and further developed in FLUIDOS.

11.1.1 KubeEdge

KubeEdge [23] is an open-source tool based on Kubernetes that enhances the orchestration capabilities for containerized applications on Edge devices. It provides basic infrastructure support for network, application deployment, and metadata synchronization between the Cloud and Edge. KubeEdge enables users to manage and monitor applications and Edge devices similar to a traditional Kubernetes cluster in the Cloud. It allows developers to write HTTP or MQTT-based applications, containerize them, and run them either at the Edge or in the Cloud, based on the specific use case.

In this context, the Cloud side is considered the master cluster, whether on-premises or managed by cloud providers like Amazon or Google. KubeEdge allows the Cloud side to connect with Edge devices remotely, deploying pods efficiently on them. To facilitate the

Funded by Horizon Europe Framework Programme of the European Union

communication, KubeEdge necessitates several core components on both Cloud and Edge sides. The most crucial components are **CloudCore** and **EdgeCore**, which will be elaborated on further in the subsequent subsections.

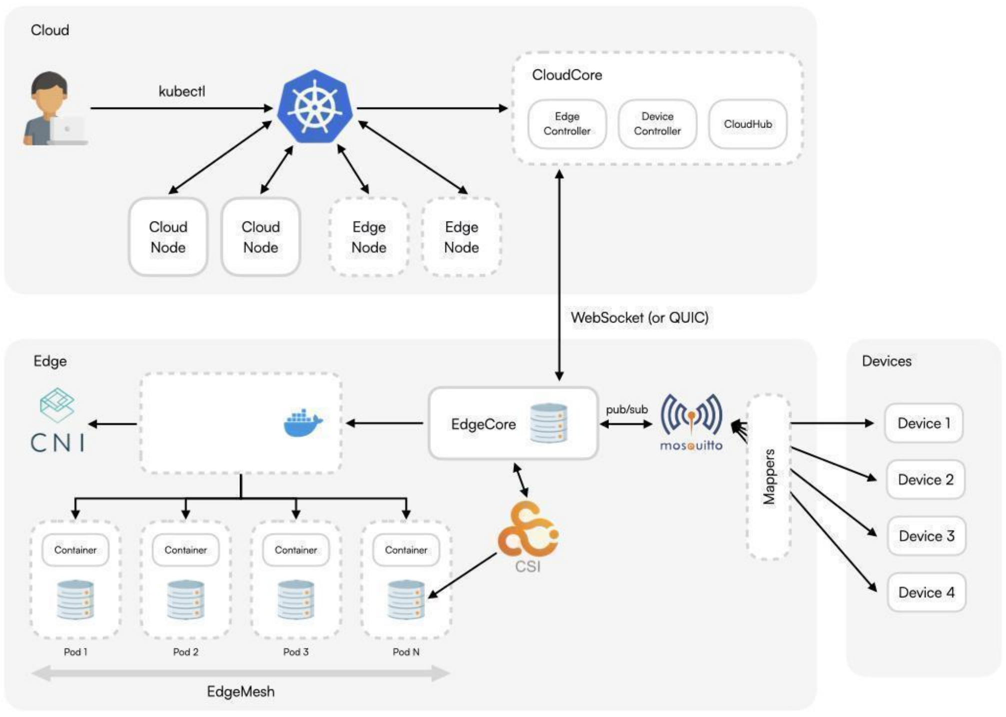


Figure 31. KubeEdge architecture.

11.1.1.1 Cloud side

To enable adding edge devices as nodes of the cluster and use them in the deployment process, the CloudCore component needs to be installed in the Kubernetes cluster on the cloud side. The CloudCore component consists of three modules: CloudHub, EdgeController, and DeviceController.

- **CloudHub** acts as a mediator between the EdgeController, DeviceController, and the Edge side. It supports both web-socket-based connections and QUIC protocol access simultaneously. The EdgeHub can select one of the protocols to access the CloudHub. CloudHub facilitates communication between the edge and the Controllers by connecting to the edge through the EdgeHub module via HTTP over the web socket connection, while for internal communication, it communicates directly with the Controllers.
- **EdgeController** serves as a bridge between Kubernetes API server and EdgeCore. It performs various functions such as syncing add/update/delete events to EdgeCore from Kubernetes API server, syncing the status of resources and events (node, pod, and configmap) to Kubernetes API server, and creating a manager interface that implements events for managing ConfigMapManager, LocationCache, and PodManager.

Funded by Horizon Europe Framework Programme of the European Union

- **DeviceController** is responsible for device management in KubeEdge. It utilizes Kubernetes Custom Resource Definitions (CRDs) to describe device metadata/status and keep this information updated between the Edge and Cloud. The device controller uses a device model and device instance to implement device management. A device model describes the properties of the devices and the related permissions required to access them, while a device instance represents an actual device object. The device spec is static, while the device status contains dynamically changing data like the desired state of a device property and the state reported by the device[8].

11.1.1.2 Edge side

The edge device requires the installation of a container runtime such as Docker or containerd, along with the EdgeCore component to communicate with the cloud. EdgeCore is made by six modules, including EdgeD and EdgeHub, with a reduced memory footprint (about 70MB). EdgeD is responsible for managing the lifecycle of pods, which enables the deployment of containerized workloads or applications on the edge node using the kubectl command line interface on the cloud side. It performs various functions, such as pod management, secret management, probe management, configmap management, container and image garbage collection, status management, volume management, and MetaClient. MetaClient is an interface of Metamanager for Edged and it helps edged to get ConfigMaps and secret details from Metamanager or the cloud. It also sends sync messages, node and pod status towards Metamanager, to the cloud.

EdgeHub serves as the communication link between the edge and cloud, connecting to the CloudHub using a web-socket connection or QUIC protocol. It ensures syncing of cloud-side resources and updating device status changes and performs functions such as keep alive, publish client info, route to cloud, and route to edge. The primary task of Route to Cloud is to collect messages from other modules and transfer them to CloudHub via a web-socket connection. On the other hand, Route to Edge is accountable for receiving messages from the cloud via the web-socket connection and sending them to the necessary groups.

[8] More information on the device model definition can be found at https://github.com/kubeedge/kubeedge/tree/master/docs/proposals/device-crd.md#device-model-type-definition.

Funded by Horizon Europe Framework Programme of the European Union

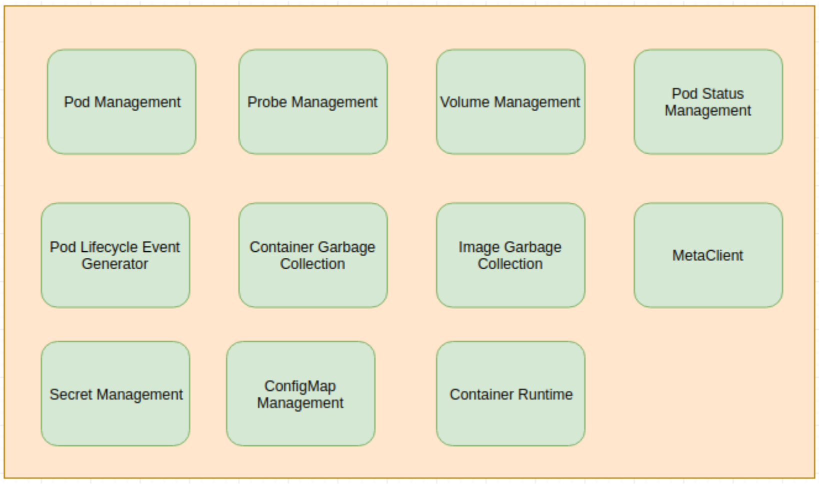


Figure 32. EdgeD functionalities.

11.1.1.3 Summary

To enable the orchestration of edge devices, it is essential to install KubeEdge components on both the cloud and edge sides. On the cloud side, a Kubernetes cluster must already be established, and CloudCore must be installed. After KubeEdge installation, the cloud side is prepared to peer with edge devices. On the edge side, it is necessary to have a container runtime. Then, the join command can be executed to install EdgeCore if it is not present and to connect the device to the Cloud side. If an edge device goes offline, the Cloud side will detect it, and the node will be labeled as NotReady. If the edge node comes back online, and Kubernetes has not marked the pods as unreachable due to the timeout, the Cloud side will detect them again.

While the most notable characteristic of KubeEdge is its reduced resource requirements on the edge nodes, it **does not provide any network, storage and service fabric** across the entire virtual infrastructure. For example, a pod running on an edge node will not belong to the same network space of pods running in the cloud, hence preventing seamless deployment of pods and services across the entire virtual space. Hence, it looks more appropriate for massive deployments of IoT-oriented applications, running at the edge, which simply gather local data and send it to other services running on the cloud side, for further processing and storage.

11.1.2 Karmada

Karmada (Kubernetes Armada) [24] is a management system for Kubernetes that allows cloud-native applications to run on multiple clusters and clouds without modifying the applications. It uses Kubernetes-native APIs and advanced scheduling capabilities to provide an open, multi-cloud Kubernetes experience. Karmada aims at automating multi-cluster application management in hybrid cloud scenarios with centralized multi-cloud management, high availability, failure recovery, and traffic scheduling. Officially, Karmada is compatible with the Kubernetes native API and it can seamlessly upgrade from single-cluster to multi-cluster while

Funded by Horizon Europe Framework Programme of the European Union

integrating with the existing K8s toolchain. However, although most of its capabilities require the use of new CRDs, hence (de facto) requiring users and dev-ops to learn a new way to interact with their clusters, as traditional Kubernetes primitives cannot be used to control the virtual continuum.

Karmada has three built-in policies for deployment scenarios and multiple scheduling policies for cluster affinity, multi-cluster splitting/rebalancing, and multi-dimension HA. It also provides centralized management for clusters in public cloud, on-premises, or at the edge.

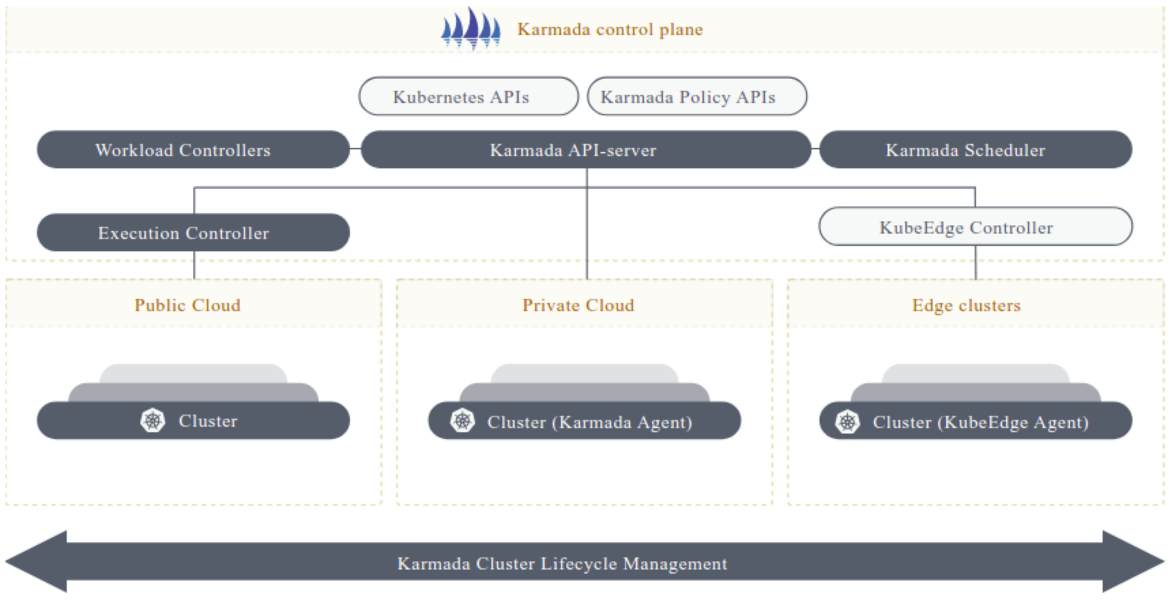


Figure 33. Karmada overview.

11.1.2.1 Concepts

Karmada has its own set of concepts and resources that manage the multi-cloud environment. These concepts include the Resource Template, which utilizes the Kubernetes Native API to create a federated resource template, allowing for seamless integration with existing Kubernetes tools. Karmada also offers the Propagation Policy API, which supports 1:n mapping of policy to workloads, allowing for easy definition of multi-cluster scheduling and spreading requirements. Users do not need to indicate scheduling constraints every time they create a federated application, as default policies enable them to interact directly with the Kubernetes API. Additionally, Karmada provides the Override Policy API, which enables users to customize the automation of cluster-related configurations, such as overriding the image prefix based on member cluster region or the StorageClass based on cloud provider. The diagram in Fig. 5 illustrates how Karmada resources propagate to member clusters.

Funded by Horizon Europe Framework Programme of the European Union

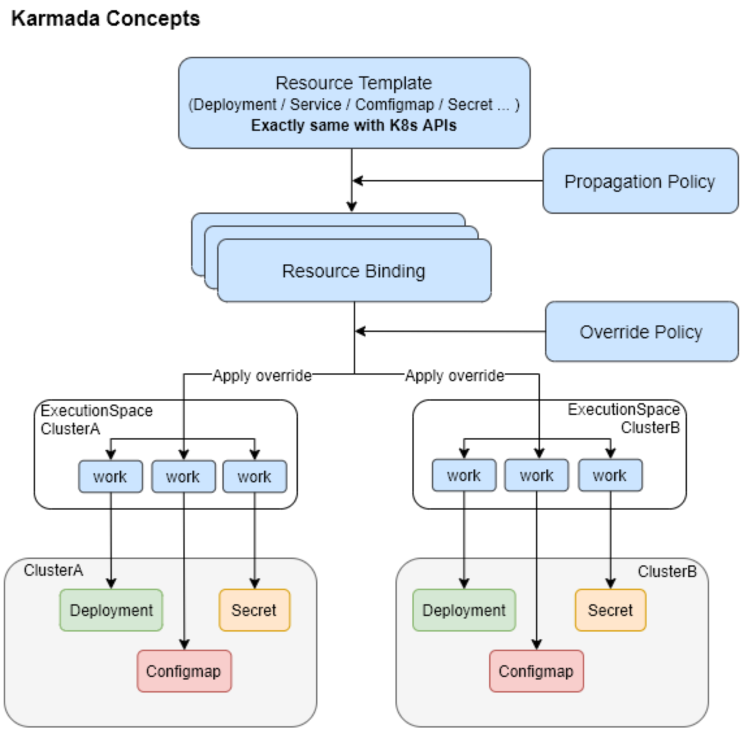


Figure 34. Karmada main concepts.

11.1.2.2 Architecture

The overall architecture of Karmada is shown in Figure 34. The Karmada Control Plane includes the Karmada API Server, Karmada Controller Manager, Karmada Scheduler, and the *etcd* database, which stores the Karmada API objects. The API server acts as the REST endpoint that all other components communicate with, and the Karmada Controller Manager carries out operations based on the API objects submitted to the API server. Internally, it runs various controllers, each of which monitors Karmada objects and takes the corresponding actions to create regular Kubernetes resources on the API servers of the underlying clusters, based on specific events.

The Cluster Controller attaches Kubernetes clusters to Karmada to manage the clusters' lifecycle by creating cluster objects. The Policy Controller watches PropagationPolicy objects and selects a group of resources that match the resourceSelector, then creates ResourceBinding with each resource object. The Binding Controller watches ResourceBinding objects and generates a Work object corresponding to each cluster with a single resource manifest. The Execution Controller watches Work objects and distributes the resources to member clusters when Work objects are created.

Funded by Horizon Europe Framework Programme of the European Union

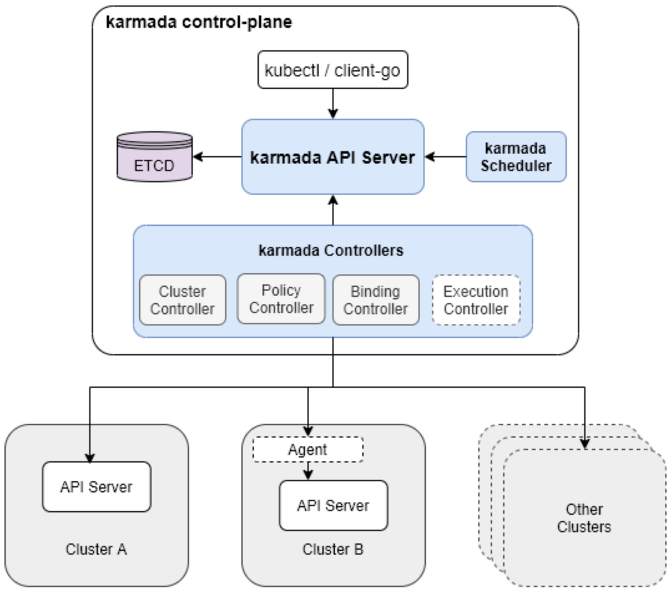


Figure 35. Karmada architecture.

11.1.2.3 Cross-cloud multi-cluster multi-mode management

Karmada offers support for safe isolation, multi-mode connection, and multi-cloud. For safe isolation, it creates a separate namespace for each cluster, which is prefixed with "karmada-es-". With multi-mode connection, Karmada can either be directly connected to the cluster kube-apiserver through a push mechanism or it can delegate tasks to an agent component installed in the cluster through a pull mechanism. Karmada also offers multi-cloud support, provided that the cloud is compliant with Kubernetes specifications. This includes support for various public cloud vendors, private cloud, and self-built clusters. The overall relationship between the member cluster and the control plane is illustrated in Figure 36.

Funded by Horizon Europe Framework Programme of the European Union

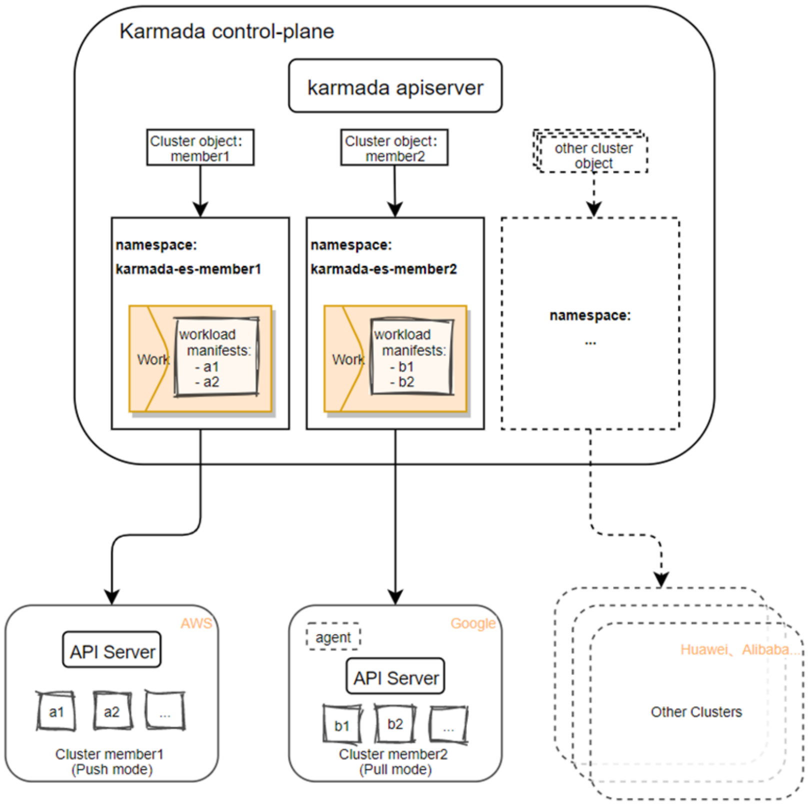


Figure 36. Karmada cross-cloud multi-mode management.

11.1.2.4 Multi-policy multi-cluster scheduling

Karmada provides support for cluster distribution capabilities under various scheduling strategies such as ClusterAffinity, Toleration, SpreadConstraint, and ReplicasScheduling. It also allows for differential configuration through OverridePolicy, which includes ImageOverrider, ArgsOverrider, CommandOverrider, and PlainText for customized differentiation configurations. Karmada further offers rescheduling functionalities with components like Descheduler and Scheduler-estimator that trigger rescheduling based on instance state changes in member clusters, providing a more precise desired state of the running instances. If a cluster lacks sufficient resources to accommodate pods, Karmada will reschedule them. The overall scheduling and rescheduling processes are illustrated in Figure 37 and Figure 38, respectively.

Funded by Horizon Europe Framework Programme of the European Union

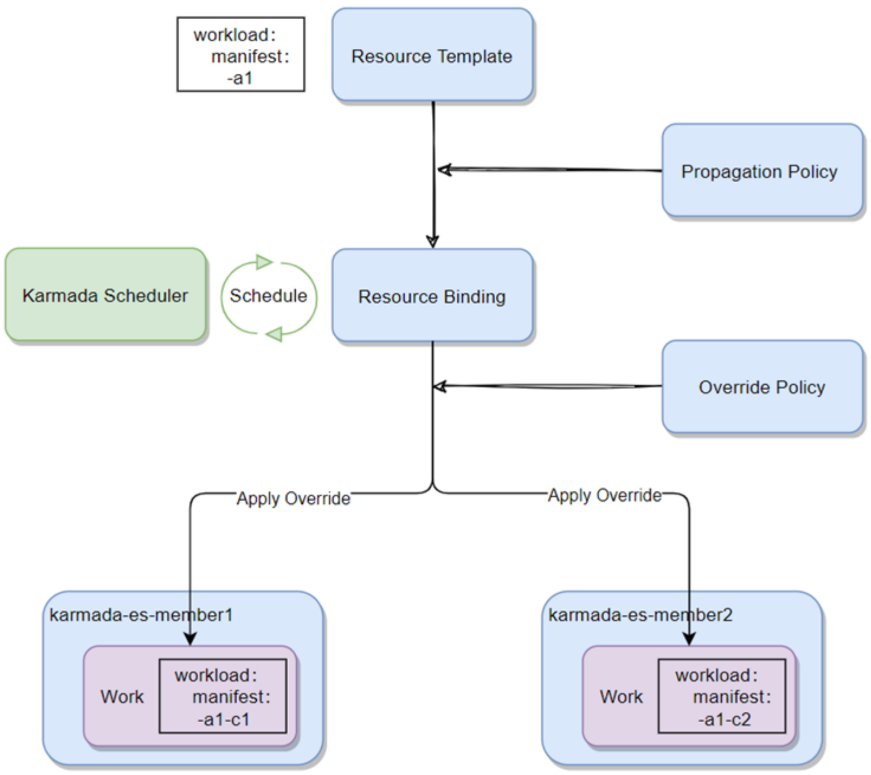


Figure 37. Karmada multi-policy multi-cluster scheduling.

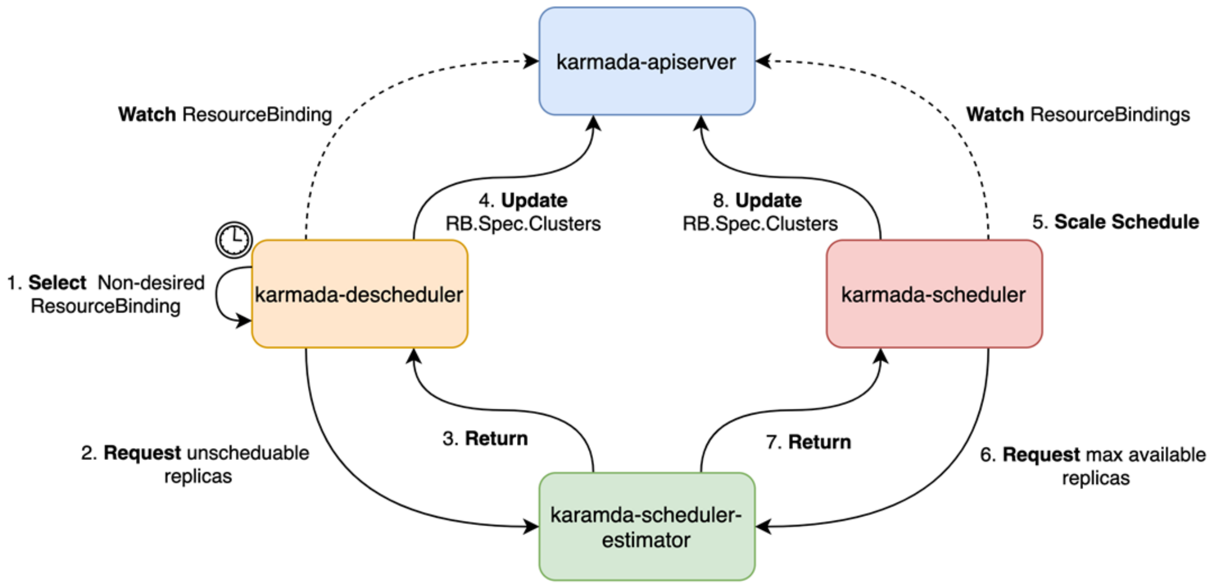

karmada-apiserver
Watch ResourceBinding
Watch ResourceBindings
4. Update RB.Spec.Clusters
8. Update RB.Spec.Clusters
5. Scale Schedule
1. Select Non-desired ResourceBinding
karmada-descheduler
karmada-scheduler
2. Request unscheduable replicas
3. Return
7. Return
6. Request max available replicas
karamda-scheduler-estimator

Figure 38. Karmada multi-policy multi-cluster re-scheduling.

Funded by Horizon Europe Framework Programme of the European Union

11.1.2.5 Cross-cluster failover of applications

Firstly, Karmada supports cluster failover (Figure 39), which involves the automatic migration of faulty cluster replicas in a centralized or decentralized manner after a cluster failure. Secondly, Karmada also supports cluster taint settings. If a user sets a taint for the cluster, and the resource distribution strategy cannot tolerate it, Karmada will trigger the migration of the cluster replicas automatically. Lastly, Karmada ensures uninterrupted service by ensuring that service replicas do not drop to zero during the migration process.

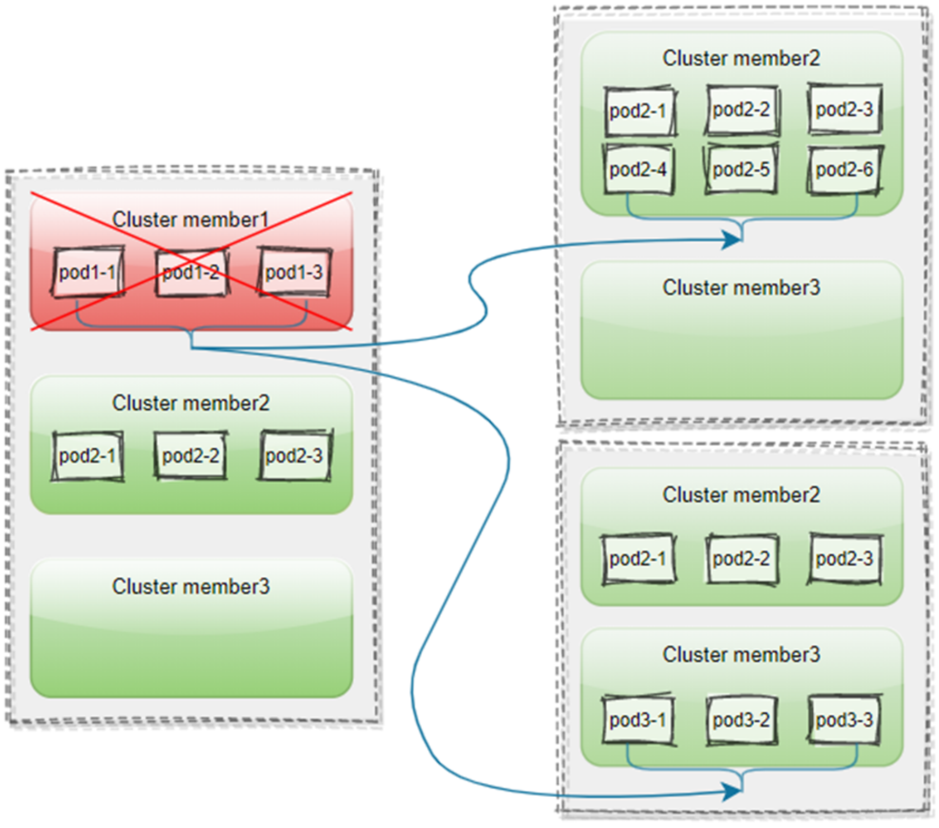


Figure 39. Karmada cross-cluster failover of applications.

11.1.2.6 Global uniform resource view

Karmada facilitates the collection and aggregation of resource status using the Resource Interpreter to create resource templates. Additionally, it enables unified management of resource creation, updating, deleting, and querying. Users can also execute operations commands, such as *describe*, *exec*, and *logs*, in one Kubernetes context. Karmada provides a global search for resources and events, including global fuzzy search and precise search through cache queries. It also supports search engines like Elasticsearch or OpenSearch, relational databases, and graph databases as third-party storage.

Funded by Horizon Europe Framework Programme of the European Union

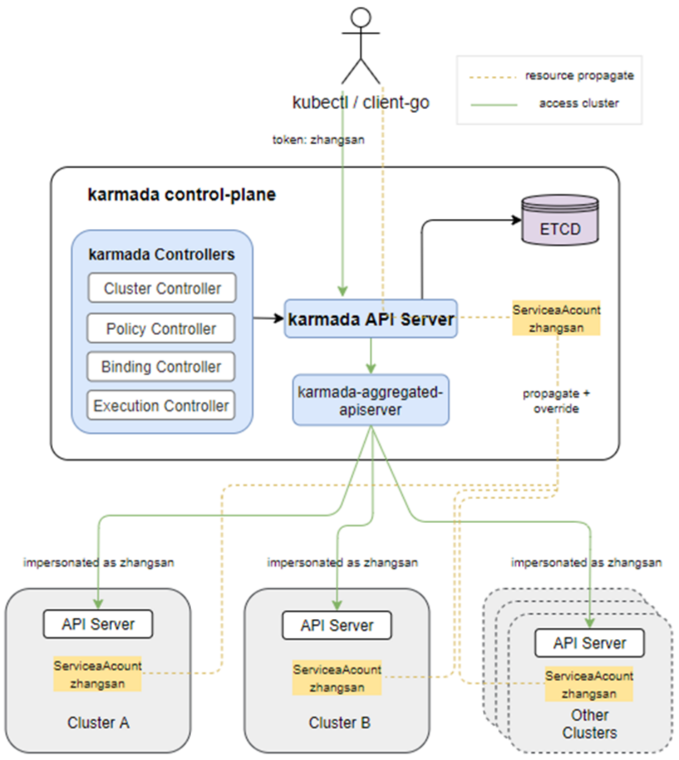


Figure 40. Access and operations via Karmada API server.

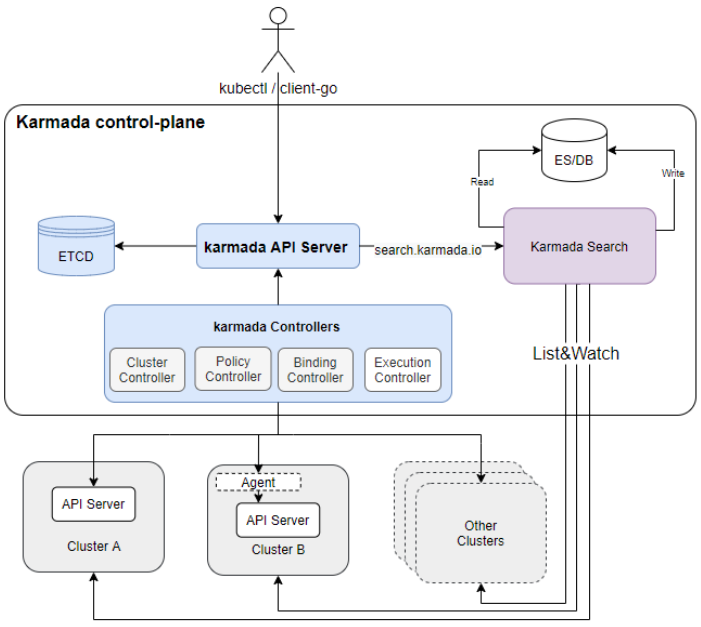


Figure 41. Checking and searching for cluster resources via Karmada API server.

Funded by Horizon Europe Framework Programme of the European Union

11.1.2.7 Access control and resource quota

Firstly, Karmada offers unified authentication with a single API access entry and access control that is consistent with member clusters. Secondly, it enables unified resource quota through FederatedResourceQuota, which allows for global configuration of resource quotas for each member cluster, federation-level configuration of resource quotas, and real-time resource usage collection for each cluster. Finally, it also supports reusable scheduling strategies by decoupling resource templates from scheduling policies, making them interchangeable.

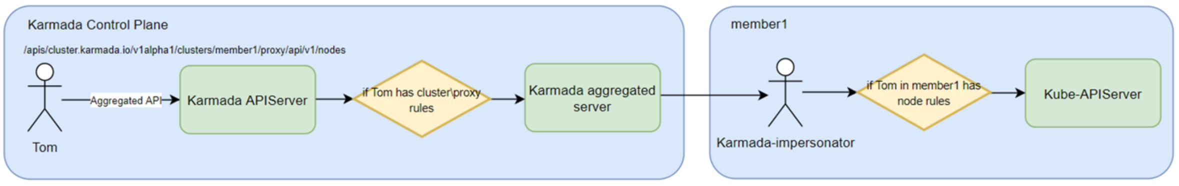

Karmada Control Plane
/apis/cluster.karmada.io/v1alpha1/clusters/member1/proxy/api/v1/nodes
Tom
Aggregated API
Karmada APIServer
if Tom has cluster\proxy rules
Karmada aggregated server
member1
Karmada-impersonator
if Tom in member1 has node rules
Kube-APIServer

Figure 42. Karmada unified authentication.

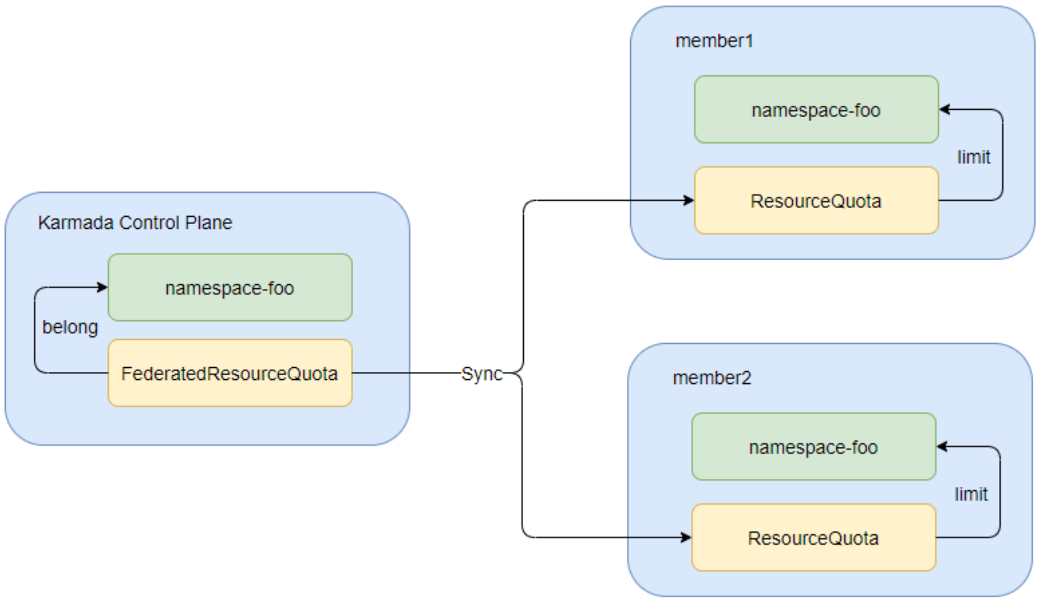


Figure 43. Defining global resource quota in Karmada.

11.1.2.8 Cross-cluster network, storage and service fabric

Karmada provides also support for multi-cluster service discovery and multi-cluster networks. With the help of ServiceExport and ServiceImport, Karmada enables cross-cluster service discovery. Additionally, Karmada leverages Submariner to connect container networks between clusters. This enables seamless communication and networking between containers in different clusters, making it easier to manage and deploy multi-cluster applications. However, it is worth nothing that this is not provided by Karmada, but it requires the usage of a different tool (Submariner, in fact), with the obvious consequences in terms of compatibility and complexity of the overall solution.

Funded by Horizon Europe Framework Programme of the European Union

Finally, Karmada provides no solutions to the creation of a storage fabric spanning across the entire virtual domain.

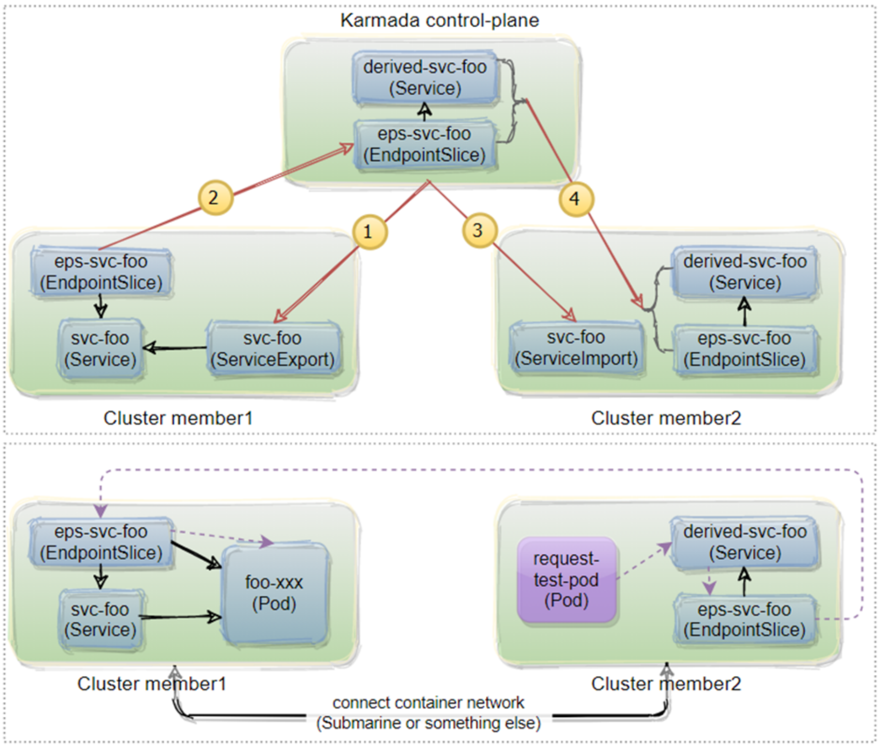

Karmada control-plane
derived-svc-foo (Service)
eps-svc-foo (EndpointSlice)
2
1
3
4
eps-svc-foo (EndpointSlice)
svc-foo (Service)
svc-foo (ServiceExport)
Cluster member1
svc-foo (ServiceImport)
derived-svc-foo (Service)
eps-svc-foo (EndpointSlice)
Cluster member2
eps-svc-foo (EndpointSlice)
svc-foo (Service)
foo-xxx (Pod)
request-test-pod (Pod)
derived-svc-foo (Service)
eps-svc-foo (EndpointSlice)
Cluster member1
Cluster member2
connect container network (Submarine or something else)

Figure 44. Service and network fabric with Karmada.

11.1.3 Liqo

Liqo [25] is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premises, cloud and edge infrastructures. While its focus is on cloud-oriented datacenter and servers, it could be used also on individual devices (e.g., coupled with lighter Kubernetes distributions, such as K3s), although it may not be appropriate to support features such as real-time scheduling and such.

Among its distinctive features, Liqo encompasses four primary capabilities, namely peering, offloading, network fabric, and storage fabric.

11.1.3.1 Peering

The process of peering in Liqo involves a one-way relationship between two Kubernetes clusters, where the consumer cluster is granted the ability to offload tasks to the provider cluster. The consumer establishes an outgoing peering towards the provider, while the provider is subjected to an incoming peering from the consumer. This setup allows for maximum flexibility in asymmetric situations and supports bidirectional peerings through their combination. Additionally, the same cluster can act as a provider and consumer in multiple peerings.

Funded by Horizon Europe Framework Programme of the European Union

To establish a peering relationship between two clusters, four tasks are involved: authentication, parameters negotiation, virtual node setup, and network fabric setup. Authentication involves pre-shared tokens to obtain a valid identity to interact with the other cluster's Kubernetes API server. The two clusters then exchange necessary parameters, including resource allocation, VPN tunnel setup, and more, and create a virtual node to abstract the resources shared by the provider cluster. The network fabric is configured to establish a secure cross-cluster VPN tunnel, enabling seamless communication between local and remote cluster pods.

Liqo supports two peering approaches, the out-of-band control plane, and the in-band control plane, which are not mutually exclusive and can be used by the same cluster for different remote clusters.

The out-of-band control plane is the standard peering method where the Liqo control plane traffic, including the initial authentication process and communication with the remote Kubernetes API server, flows outside the VPN tunnel that connects the two clusters. The VPN tunnel is established later in the peering process and is only used for cross-cluster pod traffic. However, TLS is still used to ensure secure communication.

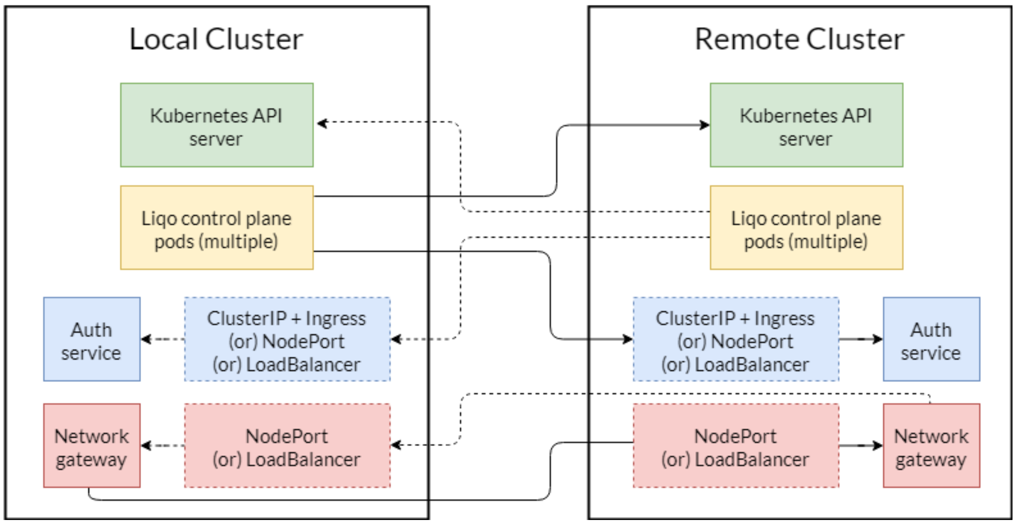

Local Cluster
Remote Cluster
Kubernetes API server
Kubernetes API server
Liqo control plane pods (multiple)
Liqo control plane pods (multiple)
Auth service
ClusterIP + Ingress (or) NodePort (or) LoadBalancer
ClusterIP + Ingress (or) NodePort (or) LoadBalancer
Auth service
Network gateway
NodePort (or) LoadBalancer
NodePort (or) LoadBalancer
Network gateway

Figure 45. Out-of-band control plane in Liqo.

On the other hand, the in-band control plane is an alternative peering method where the Liqo control plane traffic flows inside the VPN tunnel connecting the two clusters. The tunnel is established statically at the beginning of the peering process and is used for all inter-cluster traffic. The negotiation process is carried out directly by the Liqo CLI tool, and the approach requires three different cross-cluster traffic flows, as depicted in the following high-level diagram.

Funded by Horizon Europe Framework Programme of the European Union

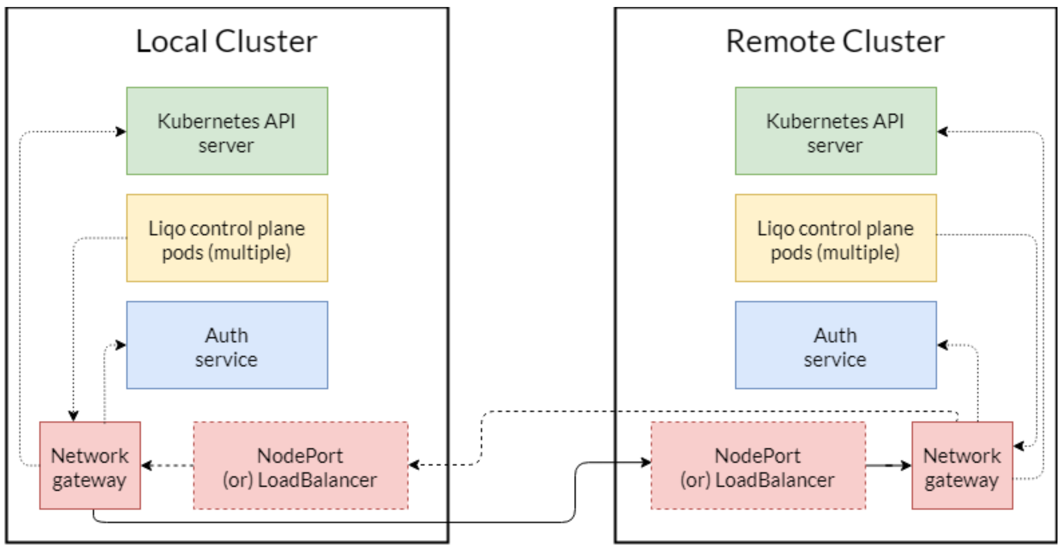


Figure 46. In-band control plane in Liqo.

11.1.3.2 Offloading

The virtual node abstraction allows for workload offloading by spawning a virtual node at the end of the local (consumer) cluster's peering process. This virtual node represents and aggregates a subset of resources shared by the remote cluster, enabling the local cluster to transparently extend its capabilities. The vanilla Kubernetes scheduler takes the new node's capabilities into account when selecting the best place for workload execution, and the approach is fully compliant with standard Kubernetes APIs, making it easy to interact with and inspect offloaded pods. The virtual node abstraction is implemented by an extended version of the Virtual Kubelet project and summarizes the resources shared by a remote cluster, automatically propagating the negotiated configuration into the capacity and allocatable entries of the node status.

To enable seamless workload offloading, Liqo extends Kubernetes namespaces across cluster boundaries, creating twin namespaces in selected remote clusters for offloaded pods. Once a pod is scheduled onto a virtual node, the corresponding Liqo virtual kubelet creates a twin-pod object in the remote cluster for actual execution. Liqo supports both stateless and stateful pods, and remote pod resiliency is ensured through a custom resource (ShadowPod) that triggers Liqo enforcement logic in the remote cluster, guaranteeing that the desired pod is always present.

The resource reflection process propagates and synchronizes selected control plane information into remote clusters, enabling the seamless execution of offloaded pods. Liqo supports the reflection of resources related to service exposition, persistent storage, and configuration data, automatically propagating resources of these types in a namespace selected for offloading into the corresponding twin namespaces created in selected remote clusters. The local copy of each resource is the source of trust for realigning the content of

Funded by Horizon Europe Framework Programme of the European Union

the shadow copy reflected remotely, with appropriate remapping of information performed by the virtual kubelet to account for conflicts and different configurations in different clusters.

11.1.3.3 Network fabric

The network fabric is the Liqo subsystem that extends the Kubernetes network model across multiple clusters, allowing pods to communicate with each other as if they were locally executed. It enables communication between pods in one cluster with those in remote peered clusters, either directly, or through the proper address space remapping (with NAT translation) in case the two clusters have overlapped network spaces. In fact, since different clusters can have varying parameters and components, it is not possible to guarantee non-overlapping pod IP address ranges. This may require address translation mechanisms, but NAT-less communication is preferred when address ranges are disjointed. The figure below depicts the network fabric established between two clusters and its main components.

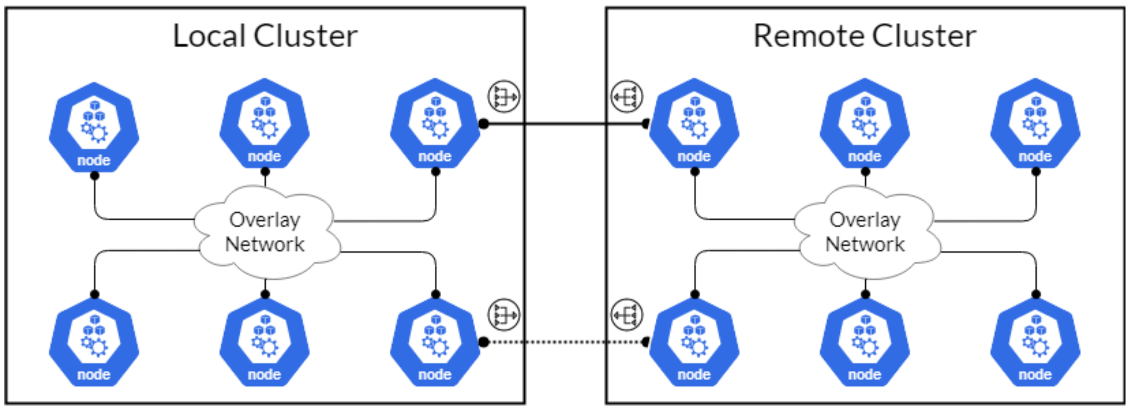


Figure 47. Liqo network fabric.

11.1.3.4 Storage fabric

The Liqo storage fabric subsystem facilitates the effortless transfer of stateful workloads to remote clusters, employing two primary methods. Firstly, storage binding is postponed until the first consumer is scheduled, either locally or remotely. Secondly, data gravity comes into play during subsequent scheduling procedures, ensuring that pods requesting existing storage pools are scheduled onto the cluster hosting the corresponding data. These techniques extend Kubernetes practice to multi-cluster scenarios, simplifying the configuration of high availability and disaster recovery scenarios. This is especially relevant for replicating and synchronizing database instances across various clusters.

Below the surface, the Liqo storage fabric uses a virtual storage class to create appropriate storage pools in different clusters. When a new PersistentVolumeClaim (PVC) linked to the virtual storage class is created, and its consumer is bound to a node (either virtual or real), Liqo's logic is activated based on the target node. In the case of a local node, PVC operations are remapped to a secondary node that corresponds to the real storage class, while for virtual nodes, the reflection logic creates a remote shadow PVC and synchronizes the

Funded by Horizon Europe Framework Programme of the European Union

PersistentVolume information. In both instances, PersistentVolumes are created with locality constraints to ensure each pod is scheduled only on the cluster with the required storage pools.

Aside from the provided storage class, Liqo also supports running pods that use cross-cluster storage managed by external solutions, such as persistent volumes provided by the cloud provider infrastructure.

11.2 RESEARCH-ORIENTED PROPOSALS

This section analyzes the solutions proposed by the research community, which includes both pure research ideas (e.g., papers without a corresponding software prototype) and proofs-of-concept or research-oriented prototypes (e.g., experimental papers). Solutions and tools presented in this section should be intended as the state of the art in literature and not production-ready solutions. Nonetheless, they can provide the ground for the future research activities in FLUIDOS project.

11.2.1 Cloudlets

Mobile cloud computing is a way to tackle the limited resources of mobile devices. It involves transferring computationally demanding tasks to the cloud. However, this approach can be ineffective for real-time applications due to the distance between the cloud and the user, resulting in high WAN latency. To address this issue, the concept of cloudlets [86] was introduced [26].

Cloudlets are trusted, resource-rich computers in the near vicinity of the mobile user (e.g., near or collocated with the wireless access point). Mobile users can then rapidly instantiate custom virtual machines (VMs) on the cloudlet running the required software in a thin client fashion.

Funded by Horizon Europe Framework Programme of the European Union

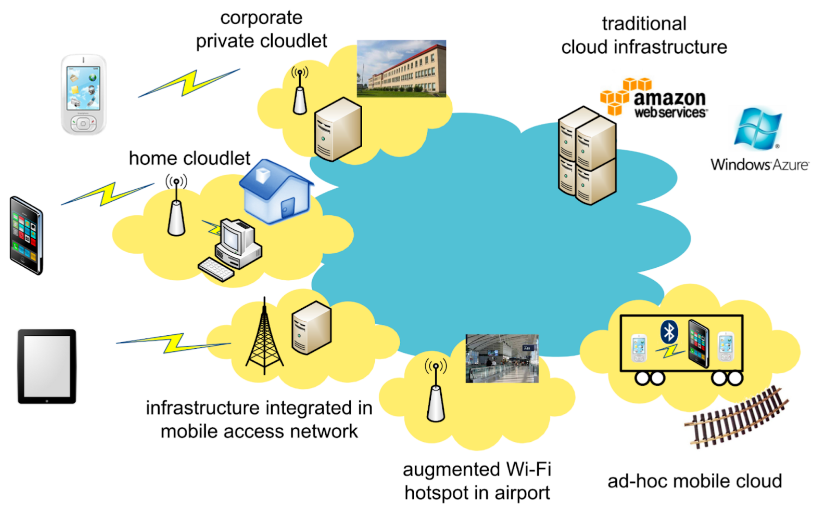

corporate private cloudlet
traditional cloud infrastructure
amazon webservices
Windows Azure
home cloudlet
infrastructure integrated in mobile access network
augmented Wi-Fi hotspot in airport
ad-hoc mobile cloud

Figure 48. Cloudnet overview.

Dynamic partitioning of an application into components can lead to improved performance. By offloading components, the allocation of cloudlet resources can be adjusted to prioritize latency-critical sections of the application, while non-real-time sections can be offloaded to more distant clouds. The cloudlet infrastructure is dynamic, allowing devices to join and leave the cloudlet during runtime.

The ad-hoc cloudlet is comprised of nodes discovered dynamically in the LAN network. These nodes run a Node Agent, which can spawn Execution Environments to deploy components. Whenever nodes join or leave the cloudlet, the Cloudlet Agent recalculates the deployments and migrates components as necessary. The elastic cloudlet operates on a virtualized infrastructure where nodes run in virtual machines. The Cloudlet Agent can spawn new nodes when additional resources are needed or stop nodes when too many resources are allocated.

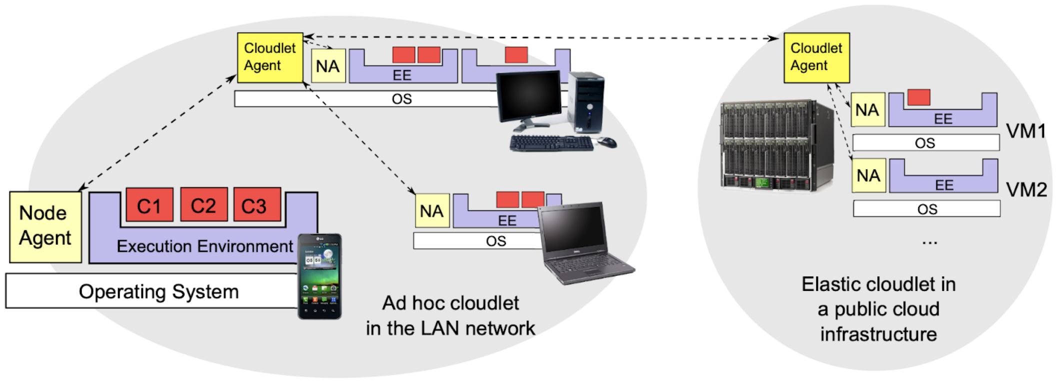


Figure 49. Cloudlets and agents.

One of the most relevant evaluation studies demonstrate that a component-based approach (the application is split in several distinct functional modules, such as micro-services) is advantageous for an Augmented Reality scenario, and the cloud is not suitable for delegating

Funded by Horizon Europe Framework Programme of the European Union

real-time constrained components. Moreover, the hierarchical design facilitates decision-making that can benefit all users and achieve a global optimum instead of conflicting local optima. Additionally, there is a need for tools to support application developers in creating component-based applications and defining quality constraints without creating additional developer burden. These tools should be able to split up applications into separate components automatically or semi-automatically.

11.2.2 FLEDGE

FLEDGE [87] is an orchestration system for containers that is compatible with Kubernetes and is specifically designed for low-resource edge devices. The system incorporates a modified Virtual Kubelets and a VPN to connect directly with Kubernetes clusters. The components of FLEDGE installed on an edge device are known as a FLEDGE agent. As FLEDGE serves as both Kubelet and container network plugin, the need for the CNI layer, which is typically situated between Kubelet and network plugin, is eliminated. Given that the number of pods that can be deployed on edge devices is limited, it is preferable to use a simple, yet effective pod networking handler (Figure 50, Container networking) that assigns IP addresses on a first come, first serve basis. The pod networking handler is also responsible for configuring the networking namespaces correctly (Figure 50, namespace manager), irrespective of the active container runtime. By labeling the node, deployment of the network plugin itself is prevented, and the system requires approximately 60MiB memory and 78MiB storage to operate on a Raspberry Pi 3, including all dependencies, which is significantly less than both Kubernetes and K3S. However, the above resource requirements are on the same order of magnitude of KubeEdge.

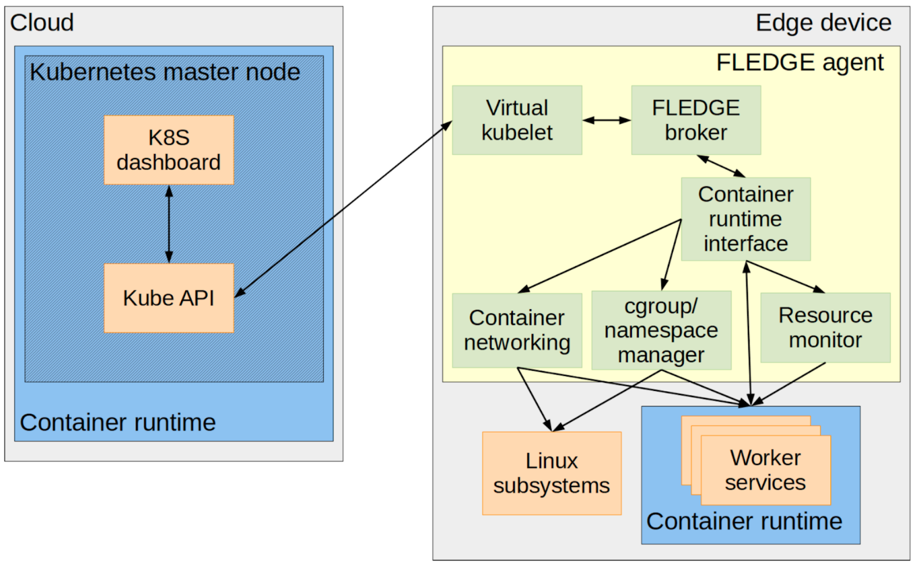


Figure 50. FLEDGE overview.

FLEDGE utilizes Kubernetes IP ranges to configure container networking, which isolates container networking within the node and does not impact other nodes in the cluster. The

Funded by Horizon Europe Framework Programme of the European Union

master node remains unaware of this approach, and the container networking plugin functions normally on other nodes. However, resource monitoring based on total resources and maximum resource allocation to pods poses challenges for edge devices. Since operating system and orchestrator resource use may represent a significant portion of total resources, it can be difficult to determine if a device can accommodate additional workloads based on pod-allocated resources alone. To address issues related to heterogeneous networks and security, FLEDGE establishes a VPN (OpenVPN) and builds the cluster and container network on top of its interface. Despite this solution, there are several downsides. First, packet encryption effectiveness depends on the chosen algorithm, and encryption can be disabled for performance reasons. Second, using a VPN negatively affects system and network resources, potentially reducing cluster scalability. Lastly, physical access to the device permits anyone to access cluster services through the VPN connection, necessitating physical and OS level security measures.

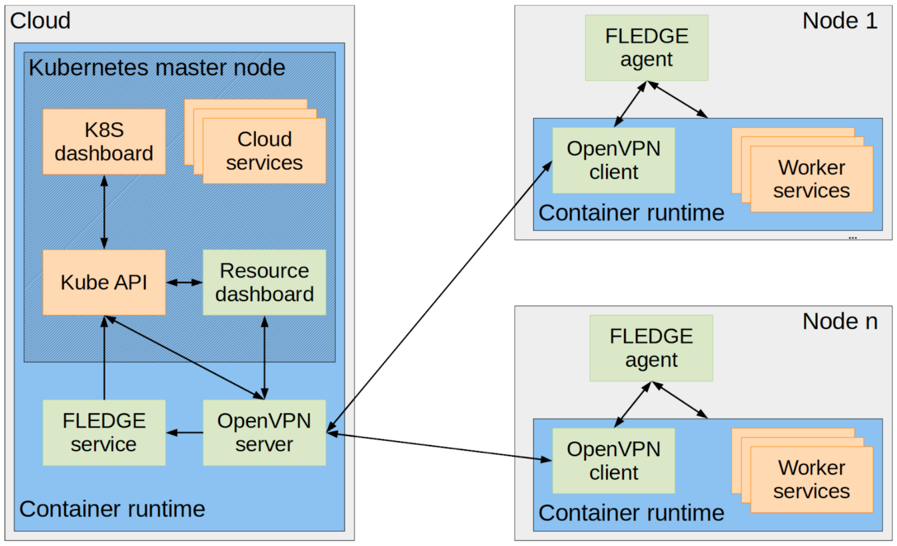


Figure 51. FLEDGE VPN.

The OpenVPN and FLEDGE containers require privileged mode access to manage network interfaces, namespaces, and cgroups. Since FLEDGE executes Kubernetes deployments, the agents must run as root, which poses a risk of unauthorized access to container images containing proprietary software. As a result, they serve as a prime attack vector, allowing access to all containers and images they manage.

The Virtual Kubelet plays a minor role in FLEDGE, primarily facilitating communication with Kubernetes master nodes. However, its location is not important since API calls can be forwarded to other devices via a custom broker implementation. There are two options for running the Virtual Kubelet depending on resource requirements:

- In the cloud, where Virtual Kubelets are executed as pods separate from FLEDGE agents, and Kubernetes API calls are forwarded to agents via REST services. This

Funded by Horizon Europe Framework Programme of the European Union

approach reduces resource demands on edge devices, making the system more robust.

- On the edge, where the Virtual Kubelet is integrated into the FLEDGE agent and executed as a container or process. API calls are performed directly in the same process, reducing operational complexity but increasing resource demands and decreasing network resilience.

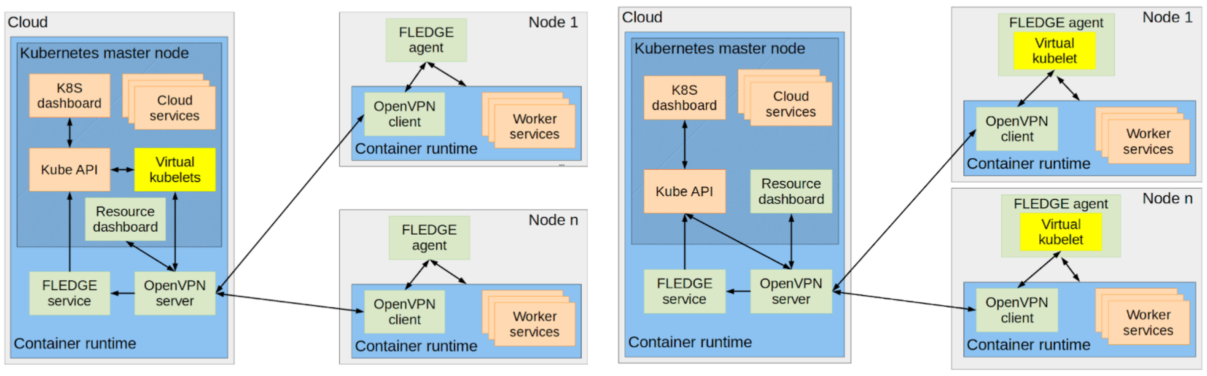


Figure 52. FLEDGE virtual kubelet.

In a preliminary evaluation, a crucial finding is that on ARM devices, a FLEDGE deployment with either Containerd or Docker requires significantly less storage than on x64. This difference is especially pronounced in the case of Docker and FLEDGE, which demands three times more storage on x64 than on ARM. The ARM versions of FLEDGE using Docker and Containerd are much more resource-efficient, consuming up to 50% less memory for Docker and 65% for Containerd. Containerd is by far the most suitable container runtime for use with FLEDGE. Overall, the ARM setup of FLEDGE using Containerd only necessitates a total of about 80MiB storage and 50MiB memory, including a VPN client container. These outcomes demonstrate that FLEDGE, which is compatible with Kubernetes, utilizes significantly fewer resources and is a feasible container orchestrator for edge devices.

11.2.3 Decentralized Kubernetes Federation Control Plane

In their position paper [88], the authors present a vision for a distributed and decentralized control plane for Kubernetes federation. The aim is to enable the support of thousands of Kubernetes clusters to cater to the needs of next-generation edge cloud use cases and, even more important, provide better and more robust support for disconnected clusters (e.g., split-brain scenario). The proposed system maintains cluster autonomy, promotes collaborative handling of error conditions among clusters, and can scale to support edge cloud use cases. The approach is based on the utilization of a shared database of conflict-free replicated data types (CRDTs) that are shared among all clusters in the federation, along with algorithms that leverage this data.

The proposed solution for a federated Kubernetes control plane involves two main tasks: configuring federated resources and propagating decisions and related data to affected

Funded by Horizon Europe Framework Programme of the European Union

clusters. Continuous operations are required for defining, configuring, and removing federated resources, and the resulting decisions and data must be distributed to the affected clusters. The proposed solution suggests replacing the central KubeFed controller with distributed federation controllers in each cluster, which apply templating, placement applicability, and value substitutions on federated resources to maintain cluster autonomy, provide resilience, and scale to meet the requirements of edge computing deployments.

To achieve distributed federation, the federated database leverages CRDTs [35], a type of distributed data that have mathematically provable conflict-free properties and can be safely distributed among nodes in a distributed database via, e.g., a Gossip protocol [36]. The proposed system extends the Kubernetes API server to separate operations on federated objects from those on standard, cluster-local objects. The federation control plane communicates indirectly through the distributed database to ensure all clusters are aware of the intended goal and current system state. Distributed federation controllers use this information to modify their local Kubernetes clusters via the API.

Due to the uncertainty around future resource availability and cluster capacity, the distributed federation controllers continuously update a shared data structure with information on how many Pods of a specific federated Deployment they have deployed locally. In situations where ideal scheduling cannot be realized due to resource unavailability, clusters can collaboratively redistribute resources among themselves. The proposed architecture uses a distributed federated database to enable distributed federation without relying on a centralized controller or a distributed etcd database.

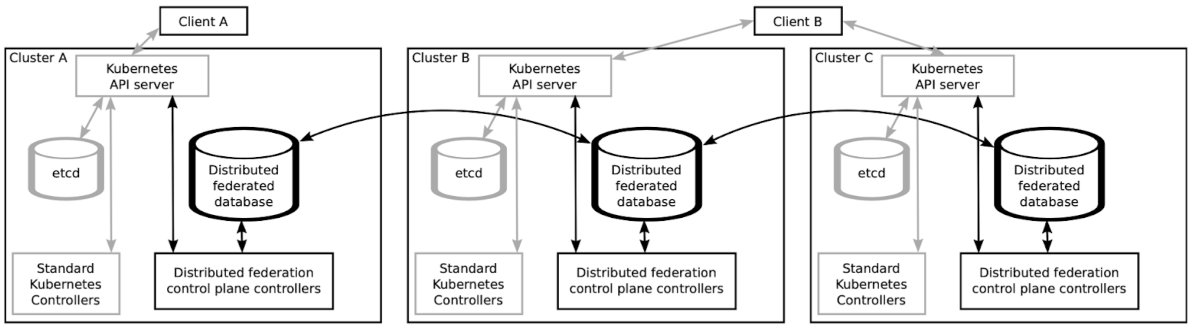

Client A
Client B
Cluster A
Cluster B
Cluster C
Kubernetes API server
etcd
Distributed federated database
Standard Kubernetes Controllers
Distributed federation control plane controllers

Figure 53. Decentralized Kubernetes federation control plane: proposed architecture.

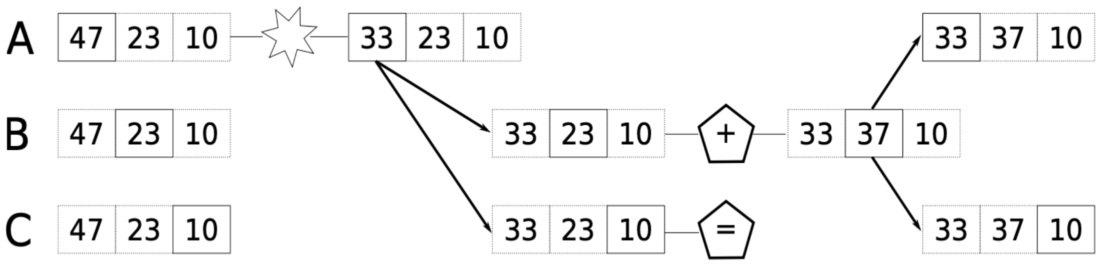


Figure 54. Decentralized Kubernetes federation control plane: resource redistribution.

Funded by Horizon Europe Framework Programme of the European Union

11.3 PAST PROJECTS

For the sake of completeness, we report here also solutions that, so far, showed little attraction or can be considered at their dead end. This would make our state-of-the-art review more complete, building on the experience (and errors) of the past projects, despite the fact they have been abandoned, deprecated, or if they just lost attention from the community.

11.3.1 KubeFed

Kubernetes Cluster Federation [28] (KubeFed for short) enables the synchronization of the settings of various Kubernetes clusters using a unified set of APIs on a host cluster. KubeFed's goal is to offer means of specifying which clusters require management of their settings and what those settings should be. KubeFed's fundamental mechanisms are purposefully simplistic and designed to serve as the groundwork for more intricate multi-cluster scenarios, such as the deployment of multi-regional applications and disaster recovery.

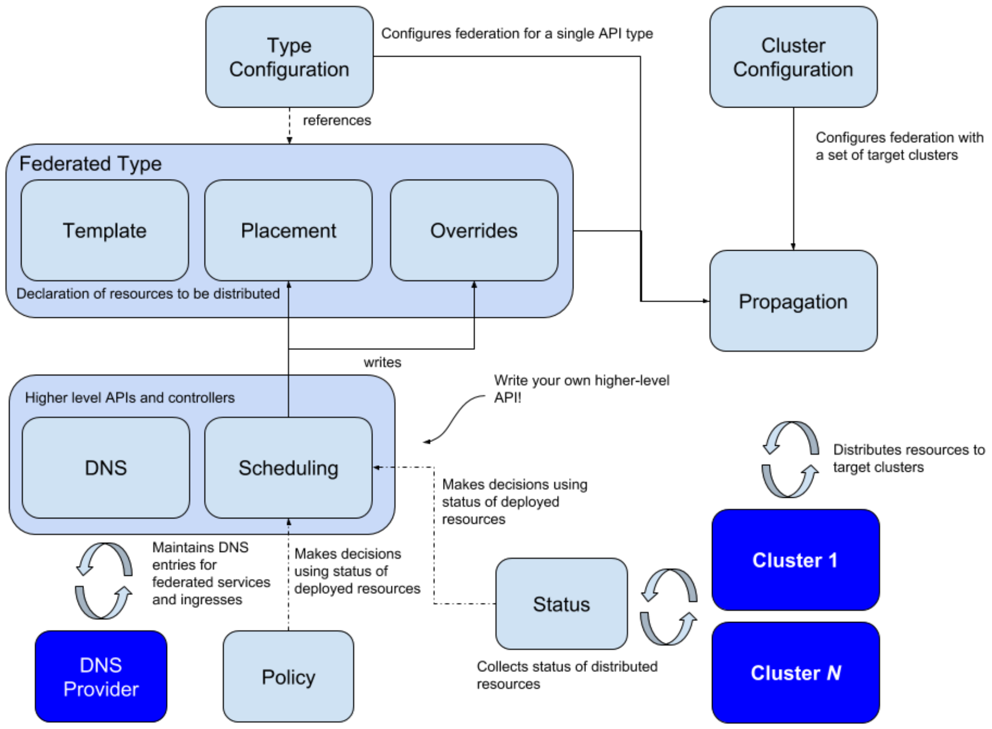

Type Configuration
Configures federation for a single API type
Cluster Configuration
references
Configures federation with a set of target clusters
Federated Type
Template
Placement
Overrides
Declaration of resources to be distributed
Propagation
writes
Higher level APIs and controllers
Write your own higher-level API!
DNS
Scheduling
Distributes resources to target clusters
Makes decisions using status of deployed resources
Maintains DNS entries for federated services and ingresses
Makes decisions using status of deployed resources
Cluster 1
Status
Cluster N
Collects status of distributed resources
DNS Provider
Policy

Figure 55. KubeFed overview.

KubeFed has two types of configurations: *type* configuration and *cluster* configuration. Type configuration specifies the API types that KubeFed should handle, while cluster configuration specifies the target clusters. Propagation is the mechanism that distributes resources to the federated clusters. Type configuration has three key concepts: templates, placement, and overrides. Templates define how a resource should look across all clusters, placement specifies which clusters a resource should appear in, and overrides allow for per-cluster

Funded by Horizon Europe Framework Programme of the European Union

variations to be applied to the template. These concepts provide the minimum information required for propagation and are useful for higher-level behaviors such as policy-based placement and dynamic scheduling. Higher-level APIs, such as Status, Policy, and Scheduling, build on these concepts to collect the status of distributed resources, determine which clusters resources are allowed to be distributed to, and make decisions about how workloads should be distributed across clusters.

After running into two different versions of the specifications (KubeFed v1, and KubeFed v2), the community mostly abandoned those efforts and considers Karmada the most promising successor to this project. It is worth mentioning that Liqo was not considered as a successor because, at the time of writing, is not officially part of the CNCF foundation.

11.3.2 Admiralty

Admiralty [29] is a system of Kubernetes controllers that efficiently organizes workloads over multiple clusters. It allows for a variety of global computing applications such as high availability, active-active disaster recovery, dynamic content delivery networks (dCDNs), distributed workflows, edge computing, Internet of Things (IoT), 5G, central access control and auditing, blue/green cluster upgrades, cluster abstraction, resource federation, cloud bursting, cloud arbitrage, and more.

The purpose of Admiralty is to link the control planes of different Kubernetes clusters. To utilize Admiralty, first, install it in each cluster that needs to be federated, and configure the clusters as sources and/or targets to build a centralized or decentralized topology. Next, annotate any pod or pod template in the source cluster, and Admiralty will transform the elected pods into proxy pods scheduled on virtual-kubelet nodes that represent the target clusters. Delegate pods in the remote clusters will run the containers, and pod dependencies and dependents will follow delegate pods, meaning they are copied as needed to target clusters. A feedback loop will update the statuses and annotations of the proxy pods to reflect those of the delegate pods, and kubectl logs and kubectl exec will work as expected. Finally, Admiralty can integrate with Admiralty Cloud/Enterprise, Cilium, and other third-party solutions to allow for networking across clusters.

To establish a connection between two clusters, one is designated as the source and the other as the target. Controllers in the source cluster, managed by the Admiralty agent, communicate with the Kubernetes API of the target cluster. The controllers in each cluster can utilize the target cluster's Kubernetes API to communicate with one another, just as they would within a single cluster.

Funded by Horizon Europe Framework Programme of the European Union

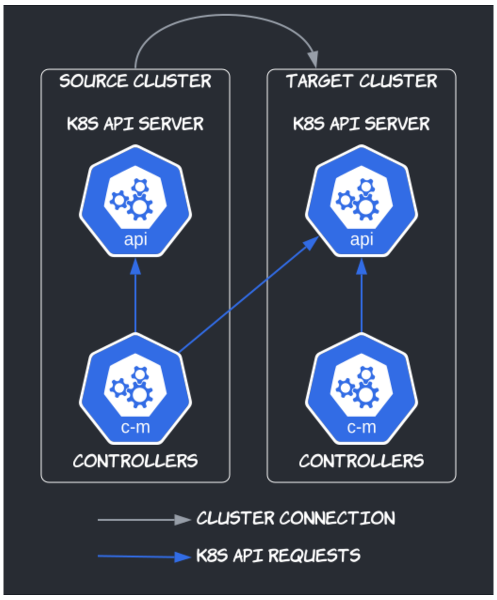


Figure 56. Cluster connections in Admiralty.

It is worth nothing that Admiralty can be considered a possible ancestor/alternative[9] of Liqo, with several ideas shared across the two project.

11.3.3 KubeML

KubeML [37] is a serverless machine learning system that is specifically designed to operate on Kubernetes and can be smoothly integrated with the widely-used PyTorch framework. This system is distinctive from other options as it fully takes advantage of GPU acceleration and can surpass TensorFlow, especially when processing smaller batches locally. When compared to TensorFlow, KubeML demonstrates a 3.98x faster time-to-accuracy with small batch sizes and maintains a 2.02x speedup for frequently benchmarked machine learning models such as ResNet34.

11.4 OTHER SOLUTIONS

For the sake of completeness, we also mention here solutions that are not directly related to the FLUIDOS project as production-ready ones, but that have some points of contact with the other projects mentioned or they are interesting for using a different approach.

[9] The word "ancestor" is not completely appropriate as the projects started, and have been developed, almost at the same time.

Funded by Horizon Europe Framework Programme of the European Union

11.4.1 EVE-OS

EVE-OS [30] is a Linux-based operating system for distributed edge computing, which is open and universal. It supports Docker containers, Kubernetes clusters, and virtual machines, offering flexibility to deploy it with any hardware, application, or cloud.

EVE-OS can be deployed on bare metal hardware such as x86, Arm, GPU or within a virtual machine to provide consistent system and orchestration services. The support for virtual machines allows users to continue to use existing software investments while building new containerized innovations simultaneously. By using bare metal EVE-OS, the possibility of bricking a device in the field during an update is eliminated, which saves the cost of an expensive truck roll.

The open EVE API is used to orchestrate the underlying hardware and installed software, enabling developers to achieve consistent behavior across a mix of technology ingredients. The project's primary focus is to maintain a state-of-the-art security posture while offering consistency and flexibility.

11.4.2 ThreeFold

The ThreeFold platform [31] operates as a decentralized peer-to-peer network that links users with local Internet resources offered by “farmers”, without the involvement of centralized servers or other intermediaries. It relies on the following two main components:

- Zero-OS (ZOS): is an operating system that is highly efficient and lightweight. Unlike other operating systems, it does not have a shell or support remote connections, except for debugging purposes. It only supports a limited number of primitives. ZOS is specifically designed by ThreeFold from the ground up to provide the necessary IT resources for solutions that run on the ThreeFold Grid. The development of ZOS began with the Linux kernel and was aimed at enhancing efficiency, energy usage, performance, scalability, management cost, and security.
- ThreeFold Grid: is a decentralized peer-to-peer Internet infrastructure that uses autonomous cloud and high-performance, permissionless blockchain technology. It is open-source and allows for the connection of compute, storage, and network capacity in one full-stack system. The ThreeFold Grid operates as a Decentralized Autonomous Organization (DAO) and consists of many servers called 3Nodes that are distributed globally by independent individuals and organizations known as ThreeFold Farmers. These 3Nodes run on a lightweight and ultra-efficient open-source operating system called Zero-OS. Each user can access the ThreeFold Grid through a Digital Self, which acts as a virtual system administrator that performs various tasks such as reserving capacity, sending messages, storing files, and building applications through a Smart Contract for IT.

Funded by Horizon Europe Framework Programme of the European Union

11.4.3 BalenaOS

BalenaOS [32] has been created to offer the essential components required for the Docker engine to work effectively in embedded scenarios. It is based on the Yocto framework and uses *systemd* as its *init* system. The networking stack consists of Network Manager, DNSmasq, and Modem Manager, which have been designed to provide a robust stack capable of handling the diverse range of hardware and unpredictable network configurations that a device may encounter during boot-up. Furthermore, BalenaOS includes Avahi, OpenSSH, and OpenVPN, which enable support for mDNS, SSH, and VPN connections respectively.

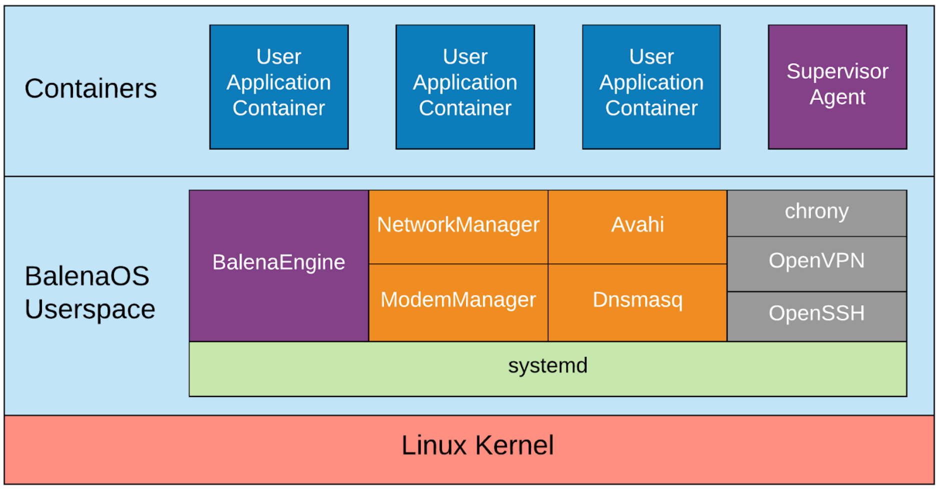


Figure 57. BalenaOS: overview.

11.4.4 Aurae

Aurae [33] is a Rust-based open-source project that offers a memory-safe systems runtime daemon called *auraed*, designed particularly for enterprise distributed systems. The auraed daemon functions as a pid 1 on a Linux kernel, overseeing containers, virtual machines, and creating short-lived nested virtual instances to add another layer of isolation.

Aurae assumes responsibility for all runtime operations on a single device and presents mTLS encrypted gRPC APIs (Aurae Standard Library) to supervise these operations. With the help of Aurae Cells, the project provides a technique for dividing a system into segments using different isolation approaches for corporate workloads.

Due to the project being in its early stages and actively developed, the APIs are frequently modified. A long-term stable release is not currently available. Therefore, it is not recommended to utilize the project in a production environment at this time.

11.4.5 Submariner

Submariner [34] enables direct networking between Pods and Services located in separate Kubernetes clusters, whether they are situated on-premises or in a cloud environment. This

Funded by Horizon Europe Framework Programme of the European Union

solution is fully open-source and has been designed to function with any network plugin (CNI). Submariner offers cross-cluster L3 connectivity through both encrypted and unencrypted connections, as well as Service Discovery across clusters. The deployment and management of this tool are made easier with the use of the *subctl* command-line utility, which also supports interconnecting clusters with overlapping CIDRs.

Submariner establishes secure and high-performance connections between multiple Kubernetes clusters. It creates a **flat network** between the clusters, enabling IP connectivity among Pods and Services. Additionally, the Lighthouse feature facilitates service discovery through the Kubernetes Multi Cluster Services model. Submariner comprises various components, including the Gateway Engine that manages secure tunnels, the Route Agent that routes cross-cluster traffic, the Broker that enables Gateway Engines to find each other, and the Service Discovery that allows DNS discovery of Services across clusters. Optional components, such as the Globalnet Controller, can provide further functionality, such as managing overlapping CIDRs between clusters. This system can work on both on-premises and public cloud environments.

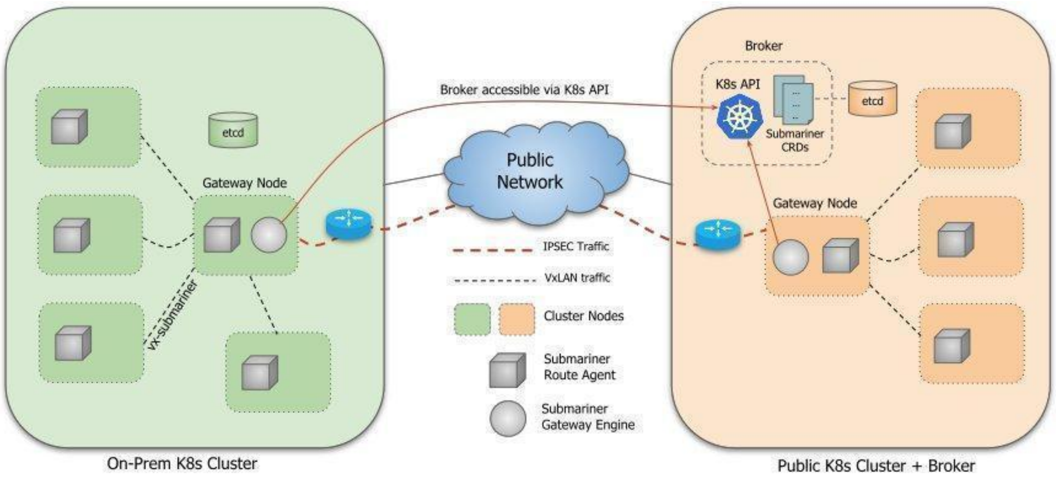


Figure 58. Submariner: overall architecture.

The Broker component acts as an intermediary between Gateway Engines located in different clusters, by exchanging metadata information. The Broker consists of Custom Resource Definitions (CRDs) that are stored in the Kubernetes datastore, along with a ServiceAccount and RBAC components to ensure secure access to its API. The Endpoint and Cluster CRDs contain information about the active Gateway Engines in a cluster and static information about the originating cluster, respectively.

The Broker is a singleton component that is deployed in a cluster whose Kubernetes API is accessible by all participating clusters. It can be deployed on a public cluster if there is a mix of on-premises and public clusters. The Gateway Engine components in each participating cluster are configured with the information to securely connect to the Broker cluster's API. The Broker cluster can be standalone without other Submariner components.

Funded by Horizon Europe Framework Programme of the European Union

During the Broker's unavailability, the data plane continues to function using the last known information, while control plane components cannot advertise or learn new or updated information from other clusters. When the connection to the Broker is restored, the components automatically synchronize their local information with the Broker and update the data plane if necessary.

The process of discovering services is facilitated by the Lighthouse project, which provides DNS discovery to Kubernetes clusters in multi-cluster environments that are connected by Submariner. This project employs the Kubernetes Multi-Cluster Service APIs to enable this functionality. Each cluster has a Lighthouse Agent that communicates with the Kubernetes API server in the Broker cluster to exchange metadata about services with other clusters. The local service information is shared with the Broker, while the service information from other clusters is imported.

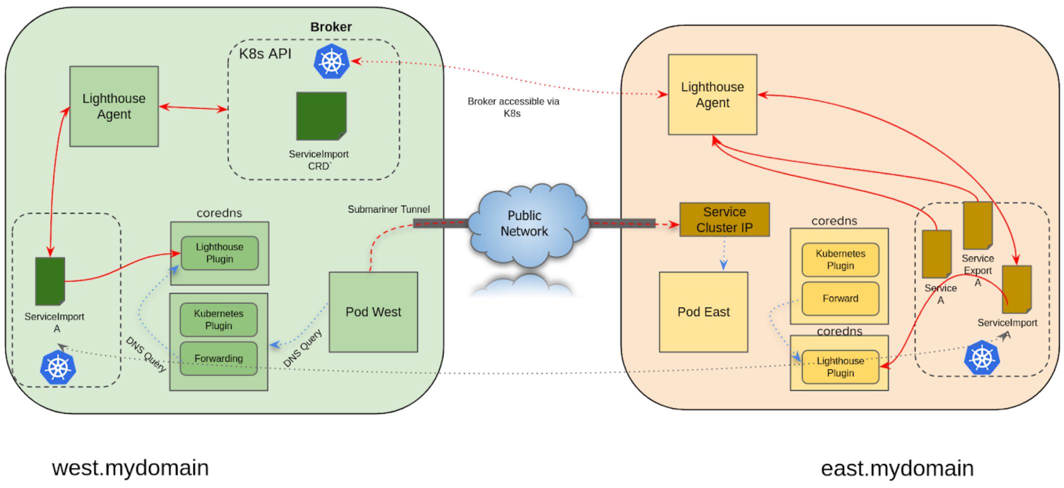


Figure 59. Submariner: basic lighthouse architecture.

Submariner has a limitation wherein it cannot handle overlapping CIDRs across clusters, which means each cluster must have a distinct CIDR that does not conflict with any other cluster. This poses a problem as most clusters use the default CIDRs, resulting in multiple clusters using the same CIDRs. Changing CIDRs requires a disruptive process and a cluster restart. To address this issue, Submariner has a component called Globalnet, which is a virtual network that supports overlapping CIDRs in connected clusters. Each cluster is given a subnet from this virtual network, and users can manually specify a GlobalCIDR for each cluster using the flag globalnet-cidr. Globalnet assigns a configurable number of global IPs per cluster, represented by ClusterGlobalEgressIP resource, which are used as egress IPs for cross-cluster communication. Users can also assign global IPs per namespace using the GlobalEgressIP resource, which takes precedence over the cluster-level global IPs. Exported ClusterIP type services are automatically allocated a global IP for ingress, while for headless services, each backing pod is allocated a global IP that is used for both ingress and egress. The routing and

Funded by Horizon Europe Framework Programme of the European Union

iptable rules use the corresponding global IPs for ingress and egress, and all address translations occur on the active Gateway node of the cluster.

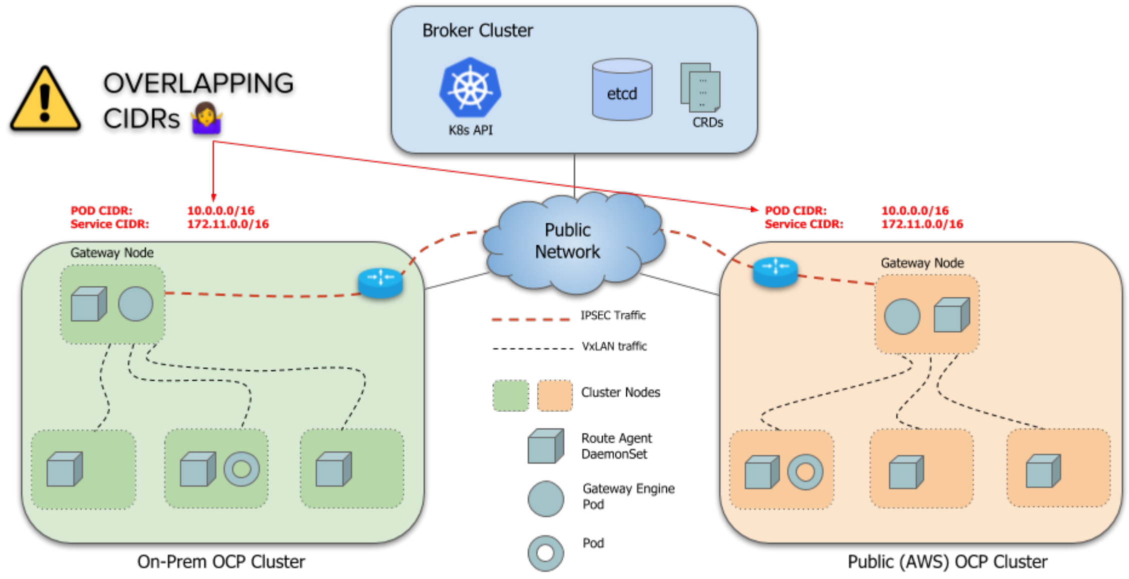


Figure 60. Submariner: overlapping IP addressing spaces (CIDR).

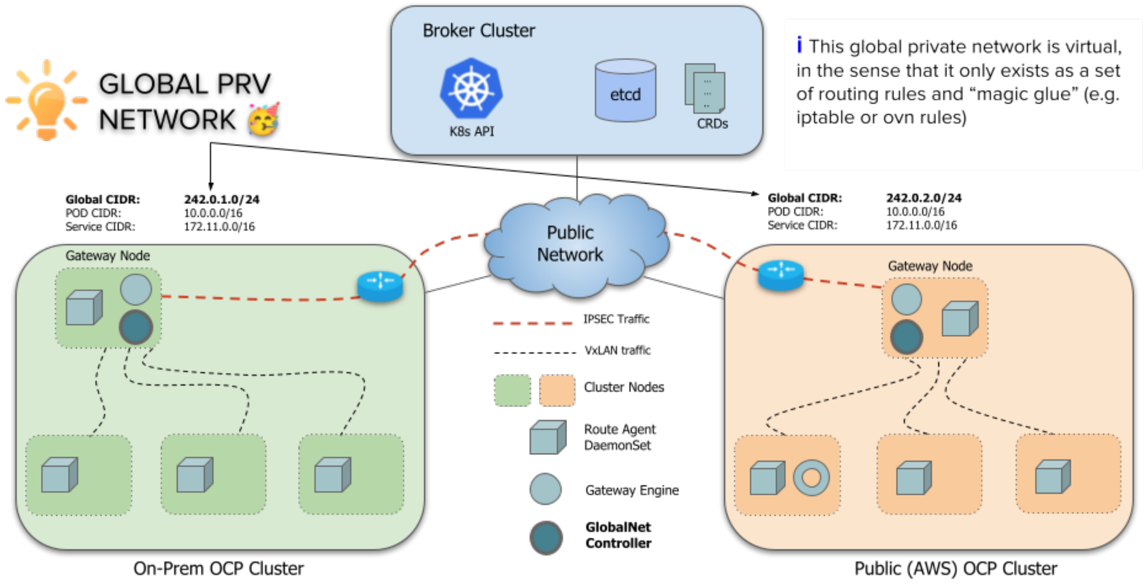


Figure 61. Submariner: global private network.

Funded by Horizon Europe Framework Programme of the European Union

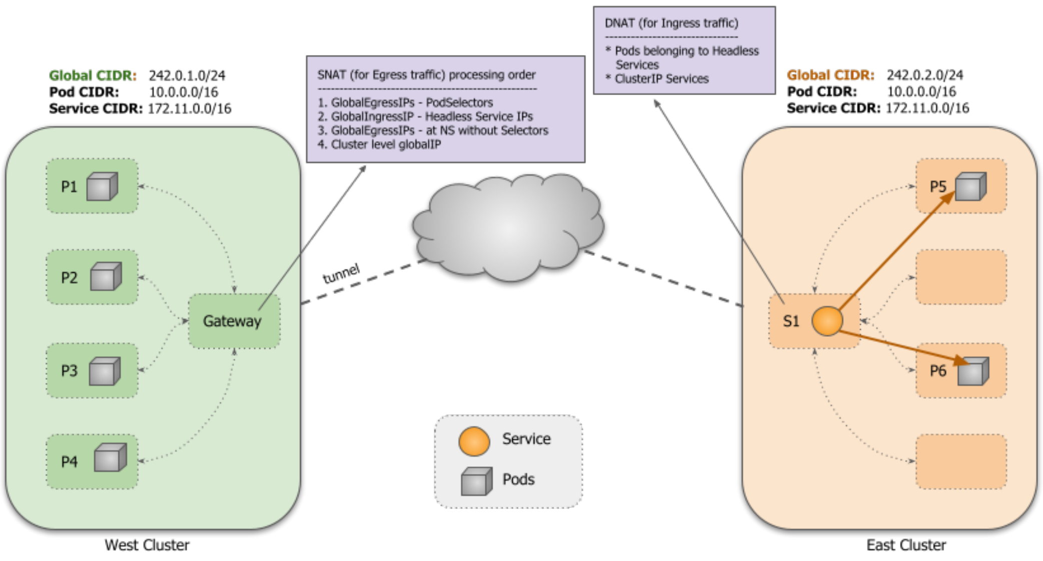

Global CIDR: 242.0.1.0/24
Pod CIDR: 10.0.0.0/16
Service CIDR: 172.11.0.0/16
SNAT (for Egress traffic) processing order
1. GlobalEgressIPs - PodSelectors
2. GlobalIngressIP - Headless Service IPs
3. GlobalEgressIPs - at NS without Selectors
4. Cluster level globalIP
DNAT (for Ingress traffic)
* Pods belonging to Headless Services
* ClusterIP Services
Global CIDR: 242.0.2.0/24
Pod CIDR: 10.0.0.0/16
Service CIDR: 172.11.0.0/16
P1
P2
P3
P4
Gateway
tunnel
S1
P5
P6
Service
Pods
West Cluster
East Cluster

Figure 62. Submariner: communications with Globalnet.

Submariner Globalnet is a tool that facilitates communication between pods and remote services using global IP addresses. It is comprised of two main components: the IP Address Manager (IPAM) and Globalnet. IPAM creates a pool of global IP addresses based on the configured GlobalCIDR, and assigns them to pods and services as needed. Globalnet sets up routing entries and iptables rules to enable cross-cluster connectivity using global IPs, and also creates additional Services with externalIPs to allow for connectivity from nodes to remote services. When a Pod, Service, or ServiceExport is deleted, the rules are removed from the gateway node. However, in addition to connectivity, pods also need to know the IPs of services on remote clusters. To address this, Lighthouse has been enhanced to support Globalnet, with the controller using a service's global IP to create ServiceImports for ClusterIP services, and the plugin using global IPs to reply to DNS queries.

With respect to FLUIDOS, Submariner creates a network fabric across clusters, but it does not provide any help for service and storage fabrics. In fact, it is used by Karmada to complement its features, but cannot be used alone to create the liquid computing vision proposed in FLUIDOS.

11.4.6 Cilium Custer Mesh

Cilium Cluster Mesh [99] is an open-source product created by Isovalent, which looks similar to Submariner in terms of overall objectives. For instance, it creates a network mesh between multiple clusters, hence enabling seamless communication between pods.

However, this solution requires that (a) all clusters are coordinated in terms of IP addresses (it does not support overlapping IP addressing space and the automatic translation of addresses through NATs); (b) all cluster must be running the Cilium CNI, hence preventing the

Funded by Horizon Europe Framework Programme of the European Union

connection of clusters running different CNIs (e.g., Cilium in one cluster, and Calico in another).

Despite the above limitations, this solution has gained a considerable attraction because of the reputation of the proposing company (Isovalent), and it is appropriate when multiple clusters are coordinated, e.g., which often happens when all the clusters are under the control of a single administrative entity.

With respect to FLUIDOS, Cilium Cluster Mesh provides a partial answer to the liquid computing problems, as it addresses only the network fabric. It is worth noting that, in the most recent versions, Cilium Cluster Mesh is compatible with Liqo, which has the capability to replace its network fabric with an external plug-in, which in fact can be Cilium Cluster Mesh.

11.4.7 VPN-based solutions

For the sake of completeness, we report here also VPN-based solutions, such as EdgeVPN [100]. The most advanced solutions in this space can create a mesh between different sites, which can be clusters in our case, hence mimicking what can be obtained with Submariner and Cilium Cluster Mesh.

With respect to FLUIDOS, the most important limitation of the above technologies is their scope, limited to the creation of a network fabric, leaving all the rest to other components.

11.4.8 Rancher Fleet

Rancher Fleet [84] is an engine for managing containers and their deployment. Its main purpose is to provide users with more control over the local cluster and constant monitoring through GitOps. Fleet is designed not to only offer scalability but also to give users a high degree of visibility and control to monitor exactly what is installed on the cluster.

Fleet can handle the management of deployments of raw Kubernetes YAML, Helm charts, Kustomize, or any combination of the three, directly from a Git repository. Regardless of the source, all resources are converted dynamically into Helm charts. Helm is used as the engine to deploy all resources in the cluster, ensuring that users have a high degree of control, consistency, and auditability.

Fundamentally, Fleet consists of a set of Kubernetes custom resource definitions (CRDs) and controllers that manage GitOps for a single Kubernetes cluster or a large-scale deployment of Kubernetes clusters. It is a distributed initialization system that makes it easy to customize applications and manage HA clusters from a single point.

Funded by Horizon Europe Framework Programme of the European Union

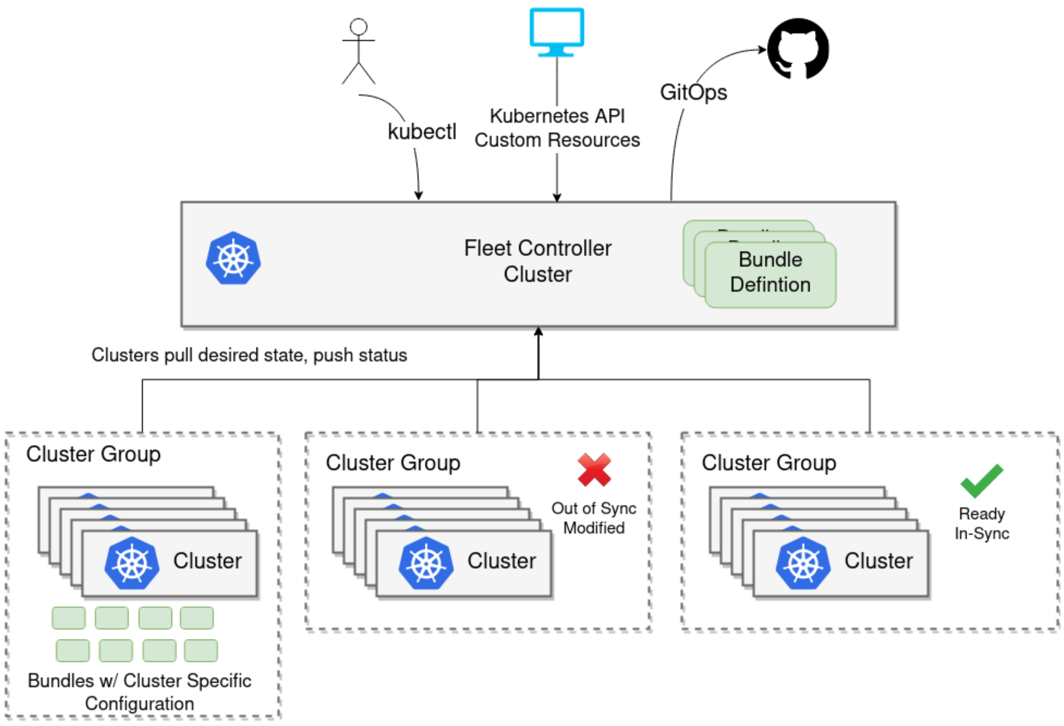


Figure 63. Rancher Fleet: architecture.

With respect to FLUIDOS, this solution helps users in the management of multiple clusters from a single interface, but it does not allow the management of multiple clusters as if they were a single one. In other words, it provides visibility across clusters, but there is no provision of a unified network, service and storage fabric: all clusters are, and remain, isolated silos.

11.4.9 Open Cluster Management

Open Cluster Management (OCM) [85] is an extensible platform designed for orchestrating Kubernetes multi-clusters. In OCM, the control plane of the multi-cluster is referred to as the *Hub*, while the individual clusters managed by the Hub are known as *Klusterlets*. The Hub Cluster refers to the cluster that runs the multi-cluster control plane, while the Klusterlets are the clusters managed by the Hub Cluster. Klusterlets are responsible for pulling the latest instructions from the Hub Cluster and consistently ensuring that the physical Kubernetes cluster is reconciled to the expected state. Typically, the Hub Cluster is a lightweight Kubernetes cluster that hosts only a few critical controllers and services.

The OCM approach separates multi-cluster operations into two parts: computation/decision and execution, according to a hub-and-spoke architecture. The hub cluster only stores the instructions for each cluster and the klusterlet agent carries out the actual execution against the target cluster. This reduces the burden on the hub cluster and makes it possible to manage large clusters. Each klusterlet works independently, so it is not reliant on the availability of the hub cluster. Even if the hub cluster becomes unreachable, the klusterlet can continue managing the hosting cluster. This is beneficial when the hub and managed clusters are

Funded by Horizon Europe Framework Programme of the European Union

owned by different admins since the klusterlet can be easily monitored and controlled. In the event of an accident, the klusterlet admin can disconnect from the hub without affecting the entire multi-cluster control plane.

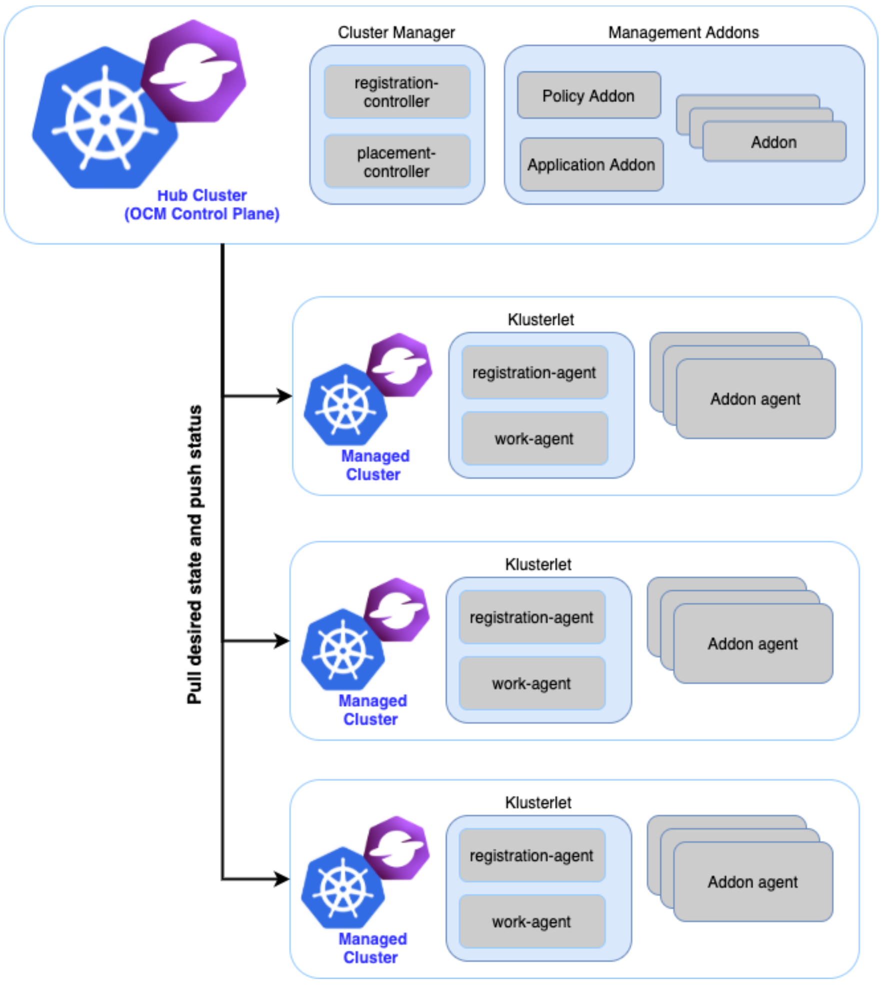


Figure 64. Open Cluster Management: architecture.

The architecture based on hub agents has the advantage of reducing the network requirements for registering a new cluster to the hub. This means that any cluster capable of accessing the hub cluster endpoint can be managed without much difficulty. This is achieved through the prescriptions being pulled from the hub rather than pushed.

With respect to FLUIDOS, this solution helps users in the management of multiple clusters from a single interface, but it does not allow the management of multiple clusters as if they were one. Thus, it provides only a partial matching of the FLUIDOS goals.

Funded by Horizon Europe Framework Programme of the European Union