UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Diseño y desarrollo de los bloques esenciales de un meta-sistema operativo para la transición hacia el paradigma del continuo de computación *Cloud-Edge-IoT*

Departamento de Comunicaciones
Universitat Politècnica de València

Tesis presentada para la obtención del grado de
Doctor por la Universitat Politècnica de València

Valencia, mayo de 2025

Autor:
Rafael Vaño Garcia

Directores:
Prof. Carlos E. Palau Salvador
Dr. Ignacio Lacalle Úbeda

Als meus pares, per la seua estima infinita.

A Jessica, per haver estat tot este temps al meu costat recolzant-me en els moments més difícils.

Si he logrado ver más lejos ha sido porque he subido a hombros de gigantes.

Resumen

El paradigma del *cloud computing* ha sido el claro dominador durante la última década, periodo en el cual numerosas corporaciones comenzaron a migrar su infraestructura de computación a la nube. El amplio dominio de este paradigma ha permitido consolidar tanto sus patrones de diseño como las tecnologías que lo habilitan (conocidas como *cloud-native*), entre las que destacan la virtualización de servicios en contenedores y su orquestación con Kubernetes.

Sin embargo, en los últimos años ha emergido una tendencia que consiste en acercar la ejecución de los servicios a las fuentes de datos, dando lugar al paradigma denominado como *edge computing*. Esta acción se traduce en diferentes mejoras, como la reducción de la latencia y del ancho de banda consumido, así como un mayor control sobre los datos, beneficios que han sido posibles gracias a una mejora en las redes de comunicaciones y al aumento paulatino de las capacidades de computación de dispositivos tradicionalmente limitados. No obstante, también introduce nuevos desafíos técnicos y operativos derivados de la heterogeneidad, la distribución y las limitaciones de estos entornos. Además, este movimiento ha ido acompañado de un esfuerzo para adaptar las tecnologías y patrones de diseño propias del *cloud* a las particularidades del *edge*, destacando nuevamente la ejecución de contenedores.

Ambos paradigmas de computación, junto con el *Internet of Things* (IoT), convergen en el llamado continuo de computación *Cloud-Edge-IoT* (CEI), que ha sido identificado como uno de los objetivos claves en materia de innovación por la Comisión Europea, y cuyo propósito consiste en la creación de un marco común de interoperabilidad sobre la heterogeneidad inherente al espectro computacional completo. Esta evolución de los paradigmas de computación ha sido analizada en esta tesis usando como hilo conductor los diferentes proyectos de investigación en los que el doctorando ha participado a lo largo de su carrera investigadora.

A pesar de estos avances, actualmente existe una brecha significativa para la implementación real del novedoso continuo de computación. Esta tesis doctoral busca contribuir a cerrar esta brecha mediante el desarrollo de los bloques elementales que habilitan la creación de un meta-sistema operativo (Meta-OS) para gobernar el continuo de computación. El desarrollo de este Meta-OS se sustenta sobre cuatro pilares claves: la federación entre dominios, la orquestación descentralizada de servicios, un sistema de monitorización y observabilidad, y una plataforma de gestión unificada.

Los componentes fundamentales de este Meta-OS han sido validados en diferentes casos de uso, que han sido seleccionados siguiendo una estrategia de validación progresiva basada en su nivel de TRL. Asimismo, el Meta-OS ha actuado como elemento habilitador clave de varios casos de uso reales pertenecientes a empresas líderes en diferentes sectores verticales, enmarcados en el proyecto europeo aerOS.

Por último, en esta tesis se aborda la prospectiva del Meta-OS desarrollado mediante la introducción de la prometedora tecnología de virtualización WebAssembly, que promete establecerse como una alternativa sólida a los contenedores en los próximos años.

Resum

El paradigma del *cloud computing* ha sigut el clar dominador durant la última dècada, període en el qual nombroses corporacions començaren a migrar la seua infraestructura de computació al núvol. L'ampli domini d'aquest paradigma ha permès consolidar tant els seus patrons de disseny com les tecnologies que l'habiliten (conegudes com a *cloud-native*), entre les que destaquen la virtualització de serveis en contenidors i la seua orquestració amb Kubernetes.

Tanmateix, en els últims anys ha emergit una tendència que consisteix en apropar l'execució de serveis a les fonts de dades, donant lloc al paradigma denominat com *edge computing*. Aquesta acció es tradueix en diferents millores, com la reducció de la latència i de l'amplada de banda consumida, així com un major control sobre les dades, beneficis que han sigut possibles gràcies a una millora en les xarxes de comunicacions y al augment progressiu de les capacitats de computació de dispositius tradicionalment limitats. No obstant, també introdueix nous reptes tècnics i operatius derivats de la heterogeneïtat, la distribució i les limitacions d'estos entorns. A més, este moviment ha anat acompanyat d'un esforç per a adaptar les tecnologies i patrons de disseny pròpies del *cloud* a les particularitats del *edge*, destacant novament l'execució de contenidors.

Ambos paradigmes de computació, junt amb el *Internet of Things* (IoT), convergeixen en el denominat continu de computació *Cloud-Edge-IoT* (CEI), que ha sigut identificat com un dels objectius claus en matèria d'innovació per la Comissió Europea, i el propòsit del qual consisteix en la creació d'un marc comú d'interoperabilitat sobre la heterogeneïtat inherent al espectre computacional complet. Aquesta evolució dels paradigmes de computació ha sigut analitzada en esta tesis utilitzant com a fil conductor els diferents projectes d'investigació en els que el doctorand ha participant al llarg de la seua carrera investigadora.

Malgrat estos avanços, actualment existeix una bretxa significativa per a la implementació real del nou continu de computació. Aquesta tesis doctoral busca contribuir a tancar esta bretxa mitjançant el desenvolupament dels blocs elementals

que habiliten la creació d'un meta-sistema operatiu (Meta-OS) per a governar el continu de computació. El desenvolupament d'este Meta-OS es sustenta sobre quatre pilars claus: la federació entre dominis, l'orquestració descentralitzada de serveis, un sistema de monitorització i observabilitat, i una plataforma de gestió unificada.

Els components fonamentals d'este Meta-OS han sigut validats en diferents casos d'ús, que han sigut seleccionats seguint una estratègia de validació progressiva basada en el seu nivell de TRL. Així mateix, el Meta-OS ha actuat com a element habilitador clau de diversos casos d'us reals que pertanyen a empreses líders en diferents sectors verticals, emmarcats dins del projecte europeu aerOS.

Per últim, en aquesta tesis s'aborda la prospectiva del Meta-OS desenvolupat mitjançant la introducció de la prometedora tecnologia de virtualització WebAssembly, que promet establir-se com una alternativa sòlida als contenidors en els pròxims anys.

Abstract

The cloud computing paradigm has been the dominant model over the past decade, during which many organizations began migrating their computing infrastructure to the cloud. This widespread adoption has led to the consolidation of both design patterns and enabling technologies (known as cloud-native), notably the virtualization of services using containers and their orchestration through Kubernetes.

In recent years, however, a trend has emerged that aims to bring service execution closer to data sources, giving rise to the paradigm known as edge computing. This shift results in several improvements, such as reduced latency, lower bandwidth consumption, and greater control over data, benefits made possible by improvements in network infrastructure and the gradual increase in computing capabilities of traditionally resource-constrained devices. Nevertheless, this approach also introduces new technical and operational challenges due to the heterogeneity, distribution, and resource constraints that characterize edge environments. In addition, this shift has been accompanied by efforts to adapt cloud-native technologies and design patterns to the specifics of the edge, with containerization once again playing a key role.

Both computing paradigms, together with the Internet of Things (IoT), converge in what is known as the Cloud-Edge-IoT continuum (CEI). This model has been identified by the European Commission as one of the key innovation priorities, with the objective of creating a common interoperability framework capable of addressing the inherent heterogeneity of the entire computing spectrum. This evolution of computing paradigms is analyzed throughout the thesis, using as a guide the various research projects in which the PhD candidate has actively been involved during his research career.

Despite recent progress, there is still a significant gap for the practical implementation of this emerging continuum. This PhD aims to address this gap through the design and development of the fundamental building blocks that enable

the creation of a meta-operating system (Meta-OS) to manage the computing continuum. The development of this Meta-OS rests on four key pillars: domain federation, decentralized service orchestration, a monitoring and observability system, and a unified management platform.

The core components of this Meta-OS have been validated in a series of use cases selected according to a progressive validation strategy based on their Technology Readiness Level (TRL). Furthermore, the developed Meta-OS has acted as the key enabling element in several real-world use cases, involving leading companies from different vertical sectors, within the scope of the European project aerOS.

Finally, the PhD thesis explores the future potential of the Meta-OS through the integration of WebAssembly, a promising virtualization technology that is expected to become a powerful alternative to containers in the coming years.

Agradecimientos

Aquesta es tracta de la secció més personal i emotiva de tota la tesis doctoral sense ningun tipus de dubte. Per este motiu, he decidit escriure esta secció d'agraïments en valencià, donat que és i sempre serà la meua llengua materna.

En primer lloc, m'agradaria donar-li les gràcies al Professor Carlos E. Palau per haver-me obert les portes de l'apassionant món de la investigació, ja que ha depositat una gran confiança en la meua persona des del primer moment. Esta etapa ha servit per a que pogués desenvolupar tot el meu potencial i convertir-me en el que sóc ara mateixa a nivell professional dins del món de les TIC.

En segon lloc, no puc oblidar-me del codirector d'esta tesis, Nacho Lacalle, ja que sense ell no haguera pogut escriure una tesis doctoral tant completa, gràcies per haver-me orientat i haver donat en el clau en els moment en els que l'escriptura semblava estancar-se. M'atreviria a dir que formem un gran equip de treball, en tant que els nostres punts forts són totalment complementaris.

També m'agradaria donar els gràcies als meus companys i excompanys del grup d'investigació SATRD per haver format part d'aquesta aventura i haver col · laborat d'alguna manera a la consecució d'esta tesis, especialment a Andreu, Álex, Benja i Paco.

Passant al plànol personal, seria una total injustícia que no me'n recordara en primer lloc dels meus pares, considerant que han fet sempre tot el possible per recolzar-me en els meus estudis, i sempre han estat i estaran al meu costat. Res d'açò haguera sigut possible sense ells, ho dic en el cor en la mà.

Què dir de tu Jessica, no saps com d'afortunat sóc per haver-te trobat durant este camí, gràcies per ser la millor companya de vida possible, ho dic en tota la sinceritat del món. Esta tesis no haguera sigut possible sense el teu recolzament i estima infinita, gràcies per haver-me suportat durant tots estos mesos inacabables d'escriptura. Encara ens queda molt de camí per recórrer junts.

També vull recordar-me de tota la meua família (tios, cosins, nebodes...), gràcies per tots els bon moments que hem passat junts, i especialment de Xaro, per haver-me guiat i donat ànims per a progressar en la meua formació acadèmica.

Per últim, este treball també està dedicat als meus amics, sou com la meua segona família ja que heu estat al meu costat quan vos he necessitat. També vull agrair-vos els moments que he passat al vostre costat, ja que han servit per a distraure'm durant l'escriptura d'esta tesis i donar-me els ànims que em faltaven per a acabar-la.

Rafa Vaño

Atzeneta, abril de 2025

Índice

Índice de figuras

Índice de tablas

Acrónimos

5G	Comunicaciones móviles de 5ª Generación
ABI	Application Binary Interface
ACK	Acknowledgement message
API	Application Programming Interface
BPF	Berkeley Packet Filter
CB	Context Broker
CI/CD	Continuous Integration and Continuous Delivery
CEI	Cloud-Edge-IoT
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNF	Cloud-Native Network Function
CNI	Container Network Interface
CP	Context Provider
CR	K8s Custom Resource
CRD	K8s Custom Resource Definition
CRI	Container Runtime Interface
CRUD	Create, Read, Update and Delete
CS	Context Source
CSI	Container Storage Interface
CSR	Context Source Registration
DevOps	Development-Operations

EB	Entrypoint balancer
EC	European Comission o Comisión Europea
eBPF	Extended Berkeley Packet Filter
EDA	Event-driven Architecture
ETSI	European Telecommunications Standards Institute
EUCEI	EUCloudEdgeIoT
gRPC	Google Remote Procedure Calls
H2020	Horizon 2020 EU's research and innovation funding programme
HA	High Availability
HE	Horizon Europe EU's research and innovation funding programme
HLO	High-Level Orchestrator
HTTP	HyperText Transfer Protocol
HTTP/2	HyperText Transfer Protocol version 2
IA	Inteligencia Artificial
IDE	Integrated Development Environment
IDL	Interface Description Language
IE	Infrastructure Element
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IIoT	Industrial Internet of Things
IoE	Internet of Everything
IoIT	Internet of Intelligent Things
IoT	Internet of Things
IP	Internet Protocol
ISG	Industry Specification Group
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
K8s	Kubernetes
LAN	Local Area Network
LB	Load Balancer o Load Balancing
LCM	Lifecycle Management
LF	Linux Foundation

LLO	Low-Level Orchestrator
MANO	Management and Orchestration
Meta-OS	Meta-Operating System
MicroVM	Micro Virtual Machine
NAT	Network Address Translation
NFV	Network Functions Virtualization
NGIoT	Next Generation Internet of Things
NGSI	Next Generation Service Interface
NGSI-LD	Next Generation Service Interface for Linked Data
OCI	Open Container Initiative
OS	Operating System
OSI	Open Systems Interconnection
OSM	Open Source NFV Management and Orchestration
P2P	Peer-to-peer
PnP	Plug-and-Play
Protobuf	Protocol Buffers
PV	Persistent Volume
PVC	Persistent Volume Claim
QUIC	Quick UDP Internet Connections
REST	Representational State Transfer
RFC	Request for Comments
RIA	Research and Innovation Action
SATRD	Sistemas y Aplicaciones de Tiempo Real Distribuidos
SBC	Single-board computer
SDK	Software Development Kit
SemVer	Semantic Versioning
SIG	Special Interest Group
SLA	Service Level Agreement
SoC	System on a Chip
SPA	Single-Page Application
SSOT	Single Source of Truth
TCP	Transfer Control Protocol
TF	Task Force
TIC	Tecnologías de la Información y la Comunicación
TLS	Transport Layer Security

UE	Unión Europea
UI	User Interface
UDP	Datagram Protocol
UPV	Universitat Politècnica de València
VM	Virtual Machine
VNF	Virtual Network Function
VPN	Virtual Private Network
VS Code	Visual Studio Code
W3C	Wide Web Consortium
WAN	Wide Area Network
WASI	WebAssembly System Interface
Wasm	WebAssembly
WG	WireGuard Working Group
WIT	Wasm Interface Type
ZT	Zero Trust

Capítulo 1

Introducción

1.1.Contexto

El Internet de las Cosas (IoT, del inglés *Internet of Things*), es decir, la conexión a la red de elementos de toda naturaleza (sensores, electrodomésticos, máquinas, vehículos...), se ha consolidado en la última década como confirman los datos relacionados con el número de dispositivos conectados a la red. Mientras que en el año 2016 el número de dispositivos conectados a nivel mundial era de aproximadamente 4.200 millones, en 2023 esta cifra alcanzó los 15.900 millones. Además, este crecimiento se espera que continúe con esta línea ascendente, puesto que se estima que para 2030 el número de dispositivos IoT conectados en todo el mundo superará los 30.000 millones [1].

Este aumento exponencial ha sido posible debido a dos motivos principalmente. El primero se trata de la mejora en las redes de comunicaciones, especialmente en las comunicaciones inalámbricas y móviles mediante el asentamiento de la tecnología 5G y su mejora constante para avanzar hacia la próxima generación 6G. Solamente en el pasado año 2024 el tráfico de datos

soportado por estas redes creció un 21%, con un promedio mensual de 157 exabytes [2]. El segundo motivo es el aumento paulatino de las capacidades de computación con las que cuentan estos dispositivos, ya que cada vez dispositivos más pequeños pueden ejecutar cargas de trabajo más demandantes, por específicas que sean.

La totalidad de estos datos ha de ser procesada en elementos de computación que cuenten con unos ciertos requerimientos de potencia para avanzar hacia la construcción de servicios o aplicaciones inteligentes. Un caso concreto se trataría de la realización análisis con técnicas *Big Data* o el entrenamiento de modelos de IA para aportar un valor añadido a los usuarios finales.

Desde la aparición y consolidación del paradigma del *cloud computing* este procesamiento ha tenido lugar en grandes centros de datos pertenecientes a empresas líderes de servicios de computación en la nube (siendo la mayoría estadounidenses como AWS, Google o Microsoft), permitiendo abstraer a las empresas y usuarios de la gestión directa de la infraestructura. Se estima que el volumen de este negocio alcanzó los 680.000 millones de dólares en 2024 habiendo crecido un 16,40% respecto al año anterior, con el consiguiente aumento necesario de las inversiones para poder seguir con el desarrollo de productos innovadores [3].

Sin embargo, recientemente ha emergido el paradigma del *edge computing*. Este paradigma consiste en trasladar paulatinamente la ejecución de estos servicios a un lugar más cercano a la fuente de los datos, es decir, los propios dispositivos IoT, con el propósito de reducir la latencia en las comunicaciones, optimizar la utilización del ancho de banda, aumentar el control sobre los datos y reducir el consumo de energía, debido a que los elementos de computación presentes en el *edge computing* cuentan con unos recursos más limitados [4].

Este movimiento ha ido acompañado de un proceso que persigue adaptar las tecnologías y patrones de diseño propias del *cloud* (*cloud-native*, como la ejecución de servicios virtualizados en forma de contenedores), que han sido probadas y validadas suficientemente ya que cuentan con un alto nivel de implementación, a los entornos propios del *edge* (*edge-native*), pero siempre teniendo en cuenta las características intrínsecas de este, como la mayor limitación de los recursos de computación, la heterogeneidad de la infraestructura o la inestabilidad de las redes [5].

El uso combinado de estos paradigmas y tecnologías de computación da lugar a un espectro o topología de computación altamente heterogéneo a varios niveles (infraestructura, red, cargas de trabajo ejecutadas...), que queda ilustrado en la Figura 1.1. Este espectro de computación abarca desde la infraestructura situada en los centros de datos de los proveedores de servicios de la nube pública, pasando por pequeños centros de datos privados o elementos con altas capacidades de computación situados en instalaciones controladas por los usuarios o las empresas, hasta finalizar en dispositivos con capacidades de computación realmente limitadas, como los sensores o actuadores IoT.

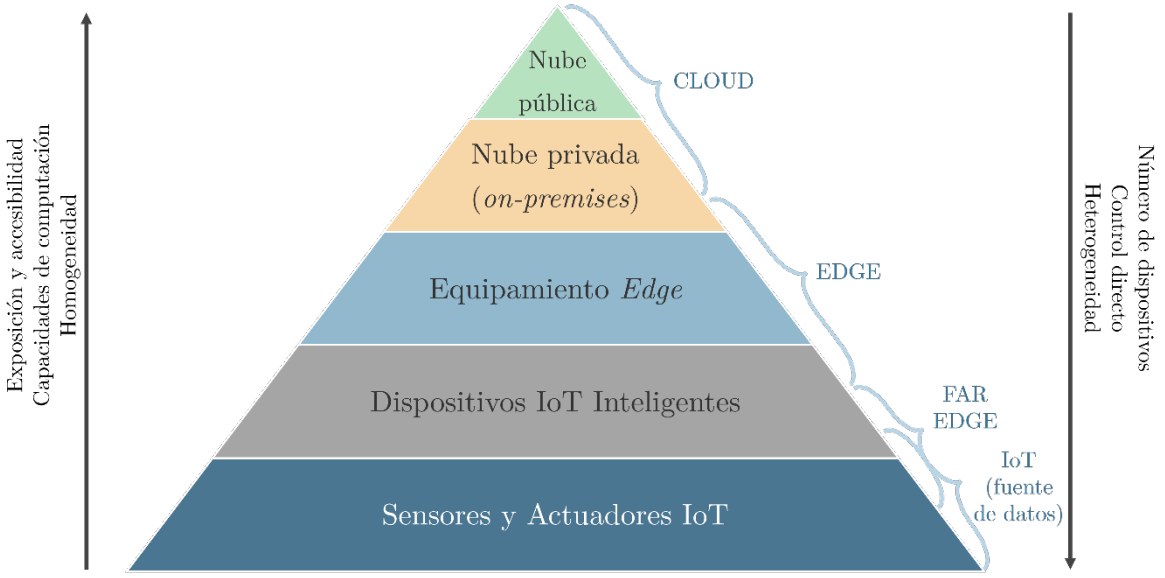


Figura 1.1: Espectro completo del continuo de computación *Cloud-Edge-IoT*

Toda esta heterogeneidad debe ser gestionada mediante la introducción de una capa de abstracción que actúe como elemento unificador y la dote de una cierta coherencia. En este ámbito aparece el concepto del continuo de computación *Cloud-Edge-IoT* (CEI), que ha suscitado un gran interés en el ámbito de las TIC, especialmente a nivel europeo con el objetivo de tomar la iniciativa en esta acción, como se puede comprobar en el lanzamiento de varias de las iniciativas de investigación y de acciones de desarrollos técnicos para abordarlo. Asimismo, los Meta-OS (meta-sistemas operativos) emergen como un elemento prometedor sobre el que construir una solución para la gestión eficiente y unificada de dicho continuo de computación, ya que estos sistemas pretenden aplicar las características y funcionalidades que aporta un OS tradicional más allá de una sola máquina, es decir, a entornos de computación distribuidos y heterogéneos.

1.2. Motivaciones

La reciente puesta en escena del paradigma del continuo de computación *Cloud-Edge-IoT* (en el que convergen la computación en la nube, el *edge computing* y los dispositivos IoT) ha abierto un campo de investigación totalmente innovador en el que tanto los líderes de los servicios de computación en la nube, como los actores que centraron sus esfuerzos en afianzarse en el tren de la innovación del *edge computing*, han puesto su foco para tratar de influenciar el desarrollo de los estándares y tecnologías sobre los que se construirá. Un ejemplo concreto es el caso de Docker y Kubernetes (impulsado inicialmente por Google), que finalmente se establecieron como los estándares para la ejecución y orquestación de contenedores, ganando la carrera a otras alternativas que se propusieron en su momento [6]. Como consecuencia, ejercieron una gran influencia en todo el ecosistema *cloud-native*, que posteriormente fue trasladada al *edge computing* [5][6].

Sin embargo, la naturaleza altamente innovadora de este paradigma se traduce en la existencia de una brecha significativa para lograr su implementación real, puesto que aún no existen soluciones ampliamente adoptadas o estandarizadas en el ecosistema tecnológico para gestionar la heterogeneidad del continuo *Cloud-Edge-IoT* [7]. Esta ausencia de estándares dificulta la optimización en el uso de los recursos computacionales que lo conforman, lo que a su vez retrasa la posibilidad de generar beneficios tangibles para los usuarios finales.

Este contexto actual converge en una serie de motivaciones, listadas a continuación, que han dado lugar a la realización de esta tesis doctoral. En particular, y como indica el título de la propia tesis, al "*Diseño y desarrollo de los bloques esenciales de un Meta-OS para la transición hacia el paradigma del continuo de computación Cloud-Edge-IoT* ":

- Contribuir al intento de la Comisión Europea de tomar el liderazgo en el ámbito del continuo *Cloud-Edge-IoT*, aprovechando la iniciativa mostrada para la mejora y adopción del *edge computing* , y así avanzar hacia la autonomía tecnológica europea en un contexto de inestabilidad mundial.
- Desarrollar un sistema que sea compatible con diversos tipos de datos existentes en importantes sectores verticales. La implementación del continuo de computación posibilitaría avanzar en el uso de estos datos de manera eficiente, traduciéndose en la mejora de los procesos tecnológicos en el ámbito industrial, y como consecuencia, en la reducción del uso recursos y del consumo de energía.
- El aumento paulatino de las capacidades de computación de los dispositivos, junto con la adaptación de las tecnologías *cloud-native* al ámbito del *edge computing*, ha permitido abrir un camino hacia la democratización de la ejecución de servicios de computación que ha de ser aprovechado para reducir la influencia de las grandes empresas tecnológicas.
- La investigación e implementación de sistemas de colaboración en entornos altamente distribuidos, en contraposición con los sistemas centralizados, con el gran desafío que esto implica.
- El desarrollo de tecnología con licencia de código abierto (*open source*) se encuentra en un momento álgido, puesto que la mayoría de la tecnología puntera en el ámbito *cloud-native* ha sido desarrollada bajo este tipo de licencias, entre las que se encuentran los proyectos bajo el paraguas de la Cloud Native Computing Foundation (CNCF), como el propio Kubernetes. Una de las principales motivaciones es hacer uso de esta tecnología para seguir contribuyendo a la apertura de la tecnología.

Asimismo, el doctorando identificó claramente la NO existencia de una serie de elementos (principalmente de carácter técnico) para avanzar hacia la implementación real del continuo de computación:

- Un modelo de datos específico para la modelación del continuo de computación, que permita reflejar el estado actual de todos los componentes que lo conforman (nodos de computación, servicios desplegados en ellos, redes que los interconectan, etc.) para dotar al continuo de una capa de observabilidad.
- Una plataforma o solución para gobernar el continuo de computación desde un punto de entrada único, que además posibilite el establecimiento de un cierto nivel de homogeneidad en un entorno esencialmente heterogéneo para abstraer a los usuarios finales de la complejidad subyacente del continuo. Esta plataforma debe poder instalarse sobre infraestructura existente y previamente configurada, sin obligar a realizar instalaciones o adaptaciones a un bajo nivel.
- Soluciones eficientes para el establecimiento de relaciones colaborativas entre nodos o dominios en entornos puramente descentralizados, que habilite el intercambio de información clave para que esta pueda ser accedida desde cualquier punto del sistema sin que se traduzca en replicaciones de la información.
- Un marco para la orquestación y despliegue de servicios de usuario (especificados siguiendo una serie de indicaciones) en sistemas de computación descentralizados, que habilite su funcionamiento con independencia del nodo de computación en que se esté ejecutando realmente.

Finalmente, es interesante destacar las motivaciones puramente académicas y del ámbito laboral que llevaron al doctorando a iniciar el proceso para realizar esta tesis doctoral

- Expandir los ámbitos de actuación y líneas de investigación del grupo de investigación SATRD (Sistemas y Aplicaciones de Tiempo Real Distribuidos) para consolidar su excelencia en la innovación.
- Consolidar sus conocimientos técnicos para su desempeño profesional (desarrollo de *software*, administración de sistemas, diseño de arquitecturas, etcétera), así como la extensión de los mismos con metodologías y tecnologías innovadoras.
- Trabajar directamente en casos de uso reales gestionados por empresas líderes pertenecientes a diferentes sectores verticales, así como el establecimiento de una relación de colaboración con los profesionales de estas empresas. Esta acción ha permitido al doctorando tener una visión más allá del ámbito académico.
- La realización de una tesis doctoral supone un esfuerzo prolongado durante varios años en el que el estudio de tecnologías y herramientas

innovadoras está siempre presente. Este esfuerzo prepara al candidato para su carrera laboral en el ámbito de las TIC, en el que el aprendizaje continuo y el reciclaje profesional tiene una presencia continua.

1.3. Objetivos de la tesis

Hipótesis

"En los últimos años se ha demostrado que las plataformas de computación se deben descentralizar para mejorar su eficiencia y seguridad, por lo que la tecnología para mover las aplicaciones a dispositivos con menor capacidad de computación ha madurado lo suficiente. Por lo tanto, es necesario alcanzar un cierto grado de homogeneidad donde se integren todas las capas del espectro de computación, y de esta manera abstraer al usuario final de la heterogeneidad del continuo de computación *Cloud-Edge-IoT*".

Objetivo general

El propósito general de esta tesis doctoral consiste en el diseño y el desarrollo de los componentes claves que habilitan un Meta-OS para gobernar el continuo de computación con la ambición de tener un enfoque altamente descentralizado, para proporcionar un marco de computación lineal y continua que abarque el espectro completo que va desde dispositivos con enorme capacidad de cómputo y de tratamiento de datos (*cloud)*, pasando por los elementos de computación menos potentes que se encuentran más cerca del usuario final y del origen de sus datos (*edge)*, hasta la misma fuente de estos datos, como los dispositivos inteligentes que forman parte del llamado *Internet of Things*. El principal objetivo es que este sistema sea capaz de abstraer al usuario final la heterogeneidad de elementos de computación y la tecnología utilizada por éstos para la ejecución de los servicios requeridos por los usuarios en forma de contenedores. Asimismo, esta tesis también persigue la validación de esta arquitectura en escenarios reales y de laboratorio, enmarcados dentro del proyecto europeo aerOS [8]. Finalmente, debido a la innovación constante que es intrínseca al sector de las TIC, como colofón final se pretende presentar un escenario de prospectiva de futuro mediante la introducción de la prometedora tecnología WebAssembly, que ha sido debidamente investigada como se puede comprobar en el estado del arte, ya que se encuentra una fase preliminar de implementación.

Objetivos específicos

Los objetivos específicos de esta tesis doctoral son los siguientes:

1) Analizar las nuevas tendencias del *Internet of Things* (incluyendo iniciativas como el Next Generation Internet of Things —NGIoT) junto a las diferentes arquitecturas IoT de referencia enfocadas al paradigma del *edge computing*.
2) Estudiar el paradigma del continuo de computación que abarca todo el espectro computacional, desde la computación en la nube hasta los dispositivos IoT: *Cloud-Edge-IoT*.

3) Desarrollar un sistema descentralizado para la gobernación del continuo de computación de una manera uniforme.
4) Explorar las diferentes tecnologías existentes para la gestión y despliegue de contenedores para su posterior integración en el sistema desarrollado.
5) Desplegar los elementos físicos y virtuales de computación (clústeres de Kubernetes, máquinas virtuales, miniordenadores, *single-board computers* —SBCs—, dispositivos IoT...) necesarios para el correcto funcionamiento de los diferentes elementos de la plataforma.
6) Asegurar la interoperabilidad del sistema con diferentes fuentes de datos.
7) Aplicar las más avanzadas técnicas de ingeniería del *software* junto con el uso de herramientas de código abierto para el diseño e implementación de los componentes necesarios del sistema o de posibles aplicaciones finales.
8) Encapsular los componentes desarrollados en contenedores junto con un sistema de orquestación y despliegue diseñado específicamente para la plataforma.
9) Utilizar dispositivos inteligentes e innovadores aplicables a la última capa de arquitectura dentro del paradigma del *edge computing*, como los que se han desarrollado dentro de proyectos de investigación como ASSIST-IoT.

1.4.Principales aportaciones

1.4.1. Artículos, Congresos y Jornadas

Artículos de revista:

- *Framework and Methodology for Establishing Port-City Policies Based on Real-Time Composite Indicators and IoT: A Practical Use-Case.* Lacalle, I.; Belsa, A.; **Vaño, R.**; Palau, C. E. Sensors 2020, 20, 4131. DOI: 10.3390/s20154131
- *Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions.* **Vaño, R.**; Lacalle, I.; Sowinski, P.; S-Julián, R.; Palau, C. E. Sensors 2023, 23(4), 2215. DOI: 10.3390/s23042215
- *Self-* Capabilities of Cloud-Edge Nodes: A Research Review.* S-Julián, R.; Lacalle, I.; **Vaño, R.**; Boronat, F.; Palau, C. E. Sensors 2023, 23(6), 2931. DOI: 10.3390/s23062931
- *A Novel Orchestrator Architecture for Deploying Virtualized Services in Next-Generation IoT Computing Ecosystems.* Mahedero Biot, F.; Fornes-Leal, A.; **Vaño, R.**; Reinosa Simón, R.; Lacalle, I.; Guardiola, C.; Palau, C.E. Sensors 2025, 25, 718. DOI: 10.3390/s25030718

Actas de congresos:

- *Leveraging IoT and prediction techniques to monitor COVID-19 restrictions in port terminals*, 2021 IEEE 7th World Forum on Internet of Things (WF-IoT), **Vaño, R.**; Lacalle, I.; Molina, B.; Palau, C. E., DOI: 10.1109/WF-IoT51360.2021.9595919
- *A Novel Approach for Calculating Real-Time Composite Indicators Relying on Internet of Things and Industrial Data Spaces*, 14th International Symposium on Intelligent Distributed Computing (IDC 2021), Belsa, A.; **Vaño, R.**; Lacalle, I.; Julián, M.; Boronat, F.; Palau, C. E., DOI: 10.1007/978-3-030-96627-0_5
- *Functioning prototype of IoT and composite indicators for smart port environmental monitoring*, IEEE International Mediterranean Conference on Communications and Networking (MeditCom 2021), Lacalle, I.; **Vaño, R.**; Palau, C. E; Milosevic, T.; Pilicic S.; Siroka, M.; Tserga, E., DOI: 10.1109/MeditCom49071.2021.9647608
- *Tactile Internet in Internet of Things Ecosystems*, 3rd International Conference on Electrical and Electronics Engineering (ICEEE 2022), Lacalle, I.; López, C.; **Vaño, R.**; Palau, C. E.; Esteve, M.; Ganzha M.; Paprzycki, M.; Szmeja, P., DOI: 10.1007/978-981-19-1677-9_69
- *Scale Model Showcase to Drive Policy Making via IoT Composite Indices and Low-Resource Equipment at the Edge of the Computing Continuum*, Second EURECA-PRO Conference on Responsible Consumption and Production 2022, **Vaño, R.**; Lacalle, I.; Palau, C. E., DOI: 10.1007/978-3-031-25840-4_52
- *Evolution of MANO Towards the Cloud-Native Paradigm for the Edge Computing*, Advanced Computing and Intelligent Technologies (ICACIT 2022), Fornés, A.; Lacalle, I.; **Vaño, R.**; Palau, C. E.; Boronat, F.; Ganzha M.; Paprzycki, M., DOI: 10.1007/978-981-19-2980-9_1
- *Design Concepts of the ASSIST-IoT Smart Orchestrator for Managing the Realization of NGIoT Systems*, 2023 IEEE 9th World Forum on Internet of Things (WF-IoT), Fornés, A. Mahedero, F.; **Vaño, R.**; Reinosa, R.; Palau, C. E.; Szmeja, P.; Lacalle, I., DOI: 10.1109/WF-IoT58464.2023.10539440
- *Autonomous Choreography of WebAssembly Workloads in the Federated Cloud-Edge-IoT Continuum*, 2023 IEEE 12th International Conference on Cloud Networking (CloudNet), Sowinski, P.; Lacalle, I.; **Vaño, R.**; Palau, C. E., DOI: 10.1109/CloudNet59005.2023.10490045
- *A Novel Approach to Self-* Capabilities in IoT Industrial Automation Computing Continuum*, 2023 IEEE 9th World Forum on Internet of Things (WF-IoT), Lacalle, I.; Cuñat, S.; Belsa, A.; **Vaño, R.**; Palau, C. E., DOI: 10.1109/WF-IoT58464.2023.10539408

- *Overview of Current Challenges in Multi-Architecture Software Engineering and a Vision for the Future,* Big Data Analytics in Astronomy, Science and Engineering (BDA 2024), Sowinski, P.; Lacalle, I.; **Vaño, R.**; Palau C.E.; Ganzha M.; Paprzycki, M., DOI: 10.1007/978-3-031-86193-2_5
- *Federation of distributed domains in the Cloud-Edge-IoT Continuum*, 2025 2nd International Workshop on MetaOS for the Cloud-Edge-IoT Continuum (MECC 2025), **Vaño, R.**; Lacalle, I.; Palau, C.E., DOI: 10.1145/3721889.3721923

1.4.2. Proyectos de investigación

El doctorando ha realizado esta tesis doctoral a partir del trabajo que ha estado realizando como personal investigador en el grupo de investigación SATRD (Sistemas y Aplicaciones de Tiempo Real Distribuidos), perteneciente al Departamento de Comunicaciones de la Universitat Politècnica de València. Por este motivo, la participación en proyectos de investigación ha jugado un papel fundamental en el programa de doctorado, puesto que han formado parte de su rutina laboral, así como de su formación en el mundo de la investigación profesional.

Estos proyectos de investigación se describirán en detalle en el capítulo 3 de esta memoria debido a la importancia de los mismos en la narrativa sobre la transición desde el surgimiento del IoT hasta su completa integración en el continuo de computación *Cloud-Edge-IoT*. En consecuencia, en este subapartado simplemente se van a realizar una enumeración introductoria de estos proyectos, aportando datos generales como su título completo y la información de su financiación.

- **PIXEL** (*Port IoT for Environmental Leverage*). Proyecto financiado por la Comisión Europea dentro del programa "Horizon 2020 (H2020)" con n.º de GA (*Grant Agreement*) 769355.
- **Plataforma Smart Terminal para Optimización de Operaciones Logísticas Portuarias VALKNUT** (*VALue Knowledge UTility*). Proyecto financiado por la Agencia Estatal de Investigación dentro del programa "Retos-colaboración 2017" con referencia RTC-2017-6566-4-AR.
- **ASSIST-IoT** (*Architecture for Scalable, Self-*, human-centric, Intelligent, Secure, and Tactile next generation IoT*). Proyecto financiado por la Comisión Europea dentro del programa "Horizon 2020 (H2020)" con n.º de GA 957258.
- **aerOS** (*Autonomous, Scalable, Trustworthy, Intelligent European Meta Operating System For The IoT-Edge-Cloud Continuum*). Proyecto financiado por la Comisión Europea dentro del programa "Horizon Europe (HE)" con n.º de GA 101069732.

1.4.3. Desarrollo software

Durante la elaboración de esta tesis doctoral, el doctorando ha realizado desarrollos de componentes *software* originales, de igual manera que ha participado en el desarrollo colaborativo de otros componentes, ya sea en su diseño, en su implementación práctica o en el testeo de nuevas funcionalidades para identificar posibles errores en su funcionamiento. Por lo tanto, en esta memoria se pueden encontrar enlaces a los repositorios públicos (mayormente apuntando a la plataforma de código pública GitHub) que contienen el código fuente de dichos componentes en la medida de lo posible, dado que el código de algunos aún no ha sido liberado de forma pública al estar inmersos en procesos de publicación más ambiciosos como parte un producto.

El doctorando es fuertemente partidario de avanzar hacia un modelo de producción de *software* y tecnología de manera abierta y colaborativa. Esta determinación se plasma en la elección de herramientas (Orion-LD, Kubernetes, containerd...), protocolos de comunicaciones y de intercambio de datos (NGSI-LD, P2P, NATS...) y lenguajes de programación (Go, JavaScript, Python...), así como en la contribución de manera activa a la mejora de todas ellas. Esta contribución se ha realizado mediante aportaciones directas de código (*Pull requests*), el reporte activo de fallos y posibles mejoras (*issues*), e incluso con la apertura de discusiones o preguntas que pueden aportar valor a los demás usuarios de la comunidad.

1.5.Organización de la memoria

La memoria de esta tesis doctoral se organiza en tres bloques diferenciados, que han sido diseñados para ilustrar las tres fases o etapas en las que se divide el desarrollo de la tesis. Cada uno de estos bloques se subdivide en diferentes capítulos que tratan diferentes aspectos específicos de la tesis con el objetivo de facilitar la lectura de la memoria, así como proporcionar una mayor información sobre el contexto en el que se engloba cada uno de ellos.

El primer bloque corresponde con la labor de investigación pura y la experiencia práctica adquirida durante este trabajo. Este bloque, que constituye el marco teórico sobre el que se fundamenta esta tesis, está formado por dos capítulos: (i) La investigación sobre **el estado del arte tecnológico**, en el que se describen los paradigmas de computación junto con sus problemáticas y desafíos, además de estándares, tecnologías y herramientas innovadoras; y (ii) La visión del doctorando sobre el proceso de **transición hacia el paradigma del continuo de computación *Cloud-Edge-IoT***, basada en la experiencia adquirida como resultado de su participación en proyectos de investigación.

El segundo bloque constituye la parte central de la tesis doctoral, puesto que abarca el proceso de diseño, implementación y validación de la solución propuesta por el doctorando al desafío planteado inicialmente. Esta parte comienza con un primer capítulo en el que se aborda **el diseño de la solución propuesta**, que se

establece en el desarrollo e implementación de los bloques básicos que habilitan la creación de un Meta-OS para la gestión del continuo de computación, mediante la descripción de los conceptos de diseño fundamentales, los requisitos técnicos y funcionales, la metodología seguida y la definición de estos bloques esenciales. Seguidamente, se afronta **la implementación de estos bloques básicos** en el capítulo más extenso de la memoria, puesto que se aportan todos los detalles técnicos de cada uno de los componentes desarrollados. Por último, este bloque finaliza con la **validación del sistema desarrollado en diferentes casos de uso**, que siguen una línea de progresión ascendente en términos de su nivel de TRL.

La memoria de la tesis termina con un bloque final de carácter puramente conclusivo y reflexivo. En primer lugar, se dedica un capítulo a la **prospectiva del Meta-OS desarrollado**, en el que se detalla la extensión del Meta-OS para el soporte de la ejecución de módulos de WebAssembly. Además, en este capítulo se exponen las líneas futuras de investigación y los trabajos futuros que se podrían llevar a cabo, esbozando las oportunidades de evolución y mejora del sistema. Por último, en el capítulo de **conclusiones** se enumeran las conclusiones finales que se extraen después de la realización de la tesis, junto con una serie de reflexiones sobre la innovación que ha aportado a este ámbito de conocimiento.

Capítulo 2

Estado del arte tecnológico

2.1.Introducción

La primera fase para realizar todo trabajo de investigación ya sea de carácter científico o social, es el análisis del estado del arte relacionado con la temática que se va a abordar para construir una base sólida sobre la que sustentar dicha investigación e intentar aportar resultados originales y enriquecedores a la comunidad.

Este capítulo se ha dividido en diferentes bloques diferenciados por la temática de estudio con el objetivo de mejorar la legibilidad y claridad de las tecnologías y conceptos expuestos. Dada su vasta extensión, se ha decidido dedicar unos párrafos para que actúe a modo de guía para lector de esta tesis doctoral, en la que se pretende realizar una breve descripción introductoria del contenido que se va a abordar en cada subsección. Es necesario destacar que gran parte de esta sección

está basada en el contenido de dos artículos de investigación escritos por el doctorando, como se muestra en la Tabla 8.1.

En primer lugar, se describe la aparición del paradigma de computación conocido como *edge computing* para enlazarlo con el nacimiento del concepto del continuo de computación *Cloud-Edge-IoT*, eje central de esta tesis. Siguiendo esta línea, se analiza el proceso actual para adaptar los conceptos de implementación y diseño propios del *cloud* computing, fuertemente establecidos y válidos, para el *edge computing*, pero siempre adaptadas a las necesidades de este último. Además, se describe como los principales proveedores de servicios de computación en la nube ofrecen soluciones propias del *edge computing* como una extensión de sus plataformas *cloud*.

Posteriormente, el foco de estudio se mueve al intercambio de datos y los protocolos de comunicaciones, empezando con un análisis sobre la gestión de la información de contexto mediante la utilización de *context brokers*. Acto seguido se realiza una comparación entre las comunicaciones síncronas y asíncronas, que incluye la enumeración de sus casos de uso típicos y de las tecnologías más innovadoras para cada uno de estos tipos.

A continuación, se introducen los desafíos y problemáticas relacionados con la heterogeneidad presente en las redes y conexiones de entornos distribuidos y del *edge computing*. Asimismo, se enumeran una serie de protocolos y tecnologías que pueden ser utilizadas para solucionarlos, haciendo una distinción en base a la capa del modelo OSI (Open Systems Interconnection, norma ISO/IEC 7498-1) [9] a la que aplican.

La siguiente sección se trata de una de las más extensas, y está dedicada por completo a la virtualización de aplicaciones mediante el uso de contenedores. En ella se explora su funcionamiento, la adaptación de esta técnica de virtualización a los entornos del *edge computing*, las herramientas disponibles para su orquestación y despliegue, el concepto *serverless* y su empaquetamiento, entre otros muchos aspectos.

El estudio del estado del arte tecnológico finaliza con una exploración de las posibles alternativas a esta tecnología de virtualización (los contenedores), haciendo hincapié en WebAssembly (junto con WASI) al ser la más prometedora y uno de los mayores vectores de innovación de esta tesis. De manera adicional, se indaga en la coexistencia de módulos de WebAssembly con contenedores en el mismo entorno de ejecución de servicios.

2.2. *Edge computing* y el continuo de computación

Las principales empresas tecnológicas han centrado sus esfuerzos a lo largo de los últimos diez a quince años en desarrollar sus sistemas y servicios de computación en la nube (*cloud computing*), paradigma en el que tanto la infraestructura como

su capacidad de computación son ofrecidas de forma remota y bajo demanda. Una característica de este paradigma es que el *hardware* que lo habilita (generalmente ubicado en grandes centros de datos) ha alcanzado un alto nivel de homogeneidad. Por lo general, todas las máquinas que forman parte de un sistema *cloud* cuentan con configuraciones uniformes, como, por ejemplo, los mismos sistemas operativos *ad hoc* que estos proveedores de servicios han personalizado para lograr un mejor rendimiento global de acuerdo con los tipos de servicios que ofrecen. Asimismo, generalmente se garantiza un alto ancho de banda y un alto grado de fiabilidad en las conexiones de red.

Sin embargo, en los últimos años se ha observado una tendencia que consiste en trasladar esta computación a entornos más locales, paradigma denominado como *edge computing*, de manera que los dispositivos locales que (generalmente) cuentan con menos recursos de computación están asumiendo más funciones de procesamiento de manera progresiva, permitiendo reducir la carga de trabajo que se realiza en la nube. Asimismo, la cercanía de estos elementos de computación a las fuentes de datos (como los dispositivos IoT) permiten reducir la latencia en las comunicaciones e incluso optimizar el ancho de banda consumido, al mismo tiempo que incrementan la seguridad e integridad de dichos datos al pertenecer a entornos más controlados. No obstante, uno de los principales inconvenientes es que el *hardware* subyacente de estos elementos es completamente heterogéneo (principalmente en arquitecturas de CPU), limitado en recursos y no está totalmente controlado por los propietarios de los sistemas. Además, están desplegados en todo tipo de entornos, incluso en entornos donde tradicionalmente no se ha desplegado infraestructura inteligente o de computación, donde la conexión de red es normalmente poco fiable [10].

Tradicionalmente, las arquitecturas del *edge computing* se han clasificado en tres categorías [11]:

- ***Multi-access edge computing*:** más bien asociado con infraestructuras de proveedores de servicios de telecomunicaciones, tiene el objetivo de trasladar los beneficios que aporta el *edge computing* a casos de uso de 4G y 5G. El Radio Access Network (RAN) y las funcionalidades *edge* se encuentran en las estaciones base de las redes de comunicaciones.
- ***Cloudlets*:** se tratan de centros de datos de tamaño reducido o de pequeñas réplicas de equipamiento de computación de la nube (*clouds in a box*) ubicados cerca del extremo de la red. Es esta ubicación de los nodos la que hace posible conseguir una reducción de la latencia, del *round-trip time* (RTT) y del ancho de banda consumido.
- ***Fog Computing* (FC):** en contraposición a los *cloudlets*, propone aprovechar la flexibilidad del IoT implementar funcionalidades propias del *edge computing*. Mediante el uso de nodos *fog*, que pueden desplegarse a través del continuo para crear múltiples capas o niveles intermedios, el FC orquesta el funcionamiento de estos nodos para

aprovechar la amplia gama de dispositivos que se encuentran en este espectro del continuo.

Se podría afirmar ésta última es la categoría que despierta más interés para el propósito de esta tesis doctoral, debido a su concepto de asociación de nodos "finales" (conocidos también como nodos del *far-edge*), ubicados cerca de los dispositivos IoT. De hecho, podrían considerarse como una primera definición de los actuales dispositivos catalogados como IoIT (*Internet of Intelligent Things*) [12]. No obstante, más allá de clasificar el *edge computing* en categorías específicas, es mucho más interesante poner énfasis en el análisis de las distintas topologías de despliegue e infraestructura. Estas topologías definen cómo interactúan entre sí los diferentes nodos que conforman un sistema de computación, puesto que la definición de tanto *edge* como *cloud* es más teórica o conceptual, ya que no existe una definición estandarizada de ambos conceptos, dando lugar a una interpretación diferente de estos términos según el contexto y el ámbito de aplicación.

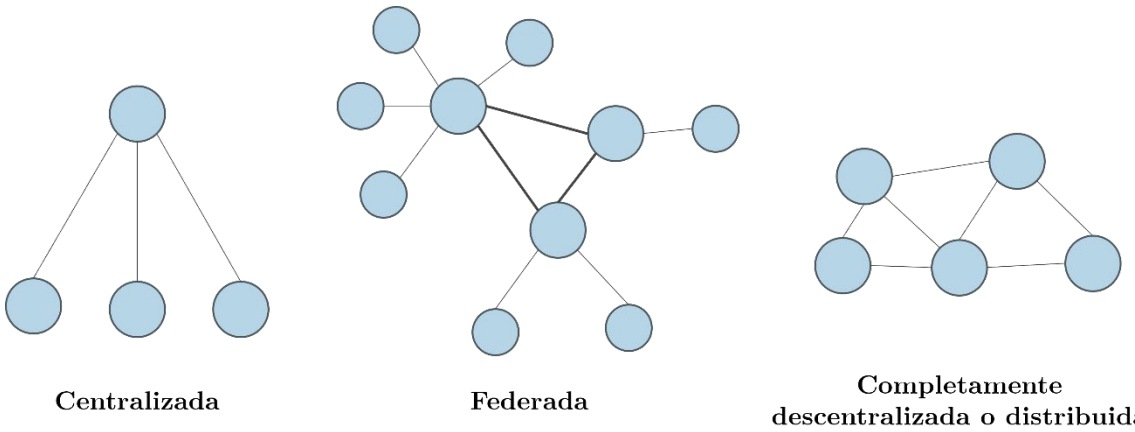


Figura 2.1: Topologías de despliegue o infraestructura

En primer lugar, en el caso de arquitecturas centralizadas, tal y como indica su propio nombre, todos los nodos dependen de un único nodo central, que suele tratarse del nodo con mayor capacidad de computación y fiabilidad (tanto de funcionamiento como de conexión de red). Esta topología se podría comparar con los sistemas *cloud* o más bien con los sistemas *cloud-edge*, en los que el nodo *cloud* coordina a los demás nodos descentralizados, que se encuentran esparcidos cerca de los usuarios o de la fuente de datos. Por otro lado, existen una serie de arquitecturas que buscan reducir la dependencia de nodos centrales, como es el caso de un continuo de computación distribuida, las cuales varían en función del grado de independencia de sus nodos. En las arquitecturas o topologías federadas se siguen manteniendo una especie de nodos centralizados en cada región o dominio independiente, encargados de coordinar a los demás nodos dentro de su dominio y, que, a su vez, interactúan y se coordinan con los nodos principales de los demás dominios. Por el contrario, en las arquitecturas totalmente descentralizadas o distribuidas, no existe una jerarquía explícita entre nodos, es decir, no se establece ninguna distinción entre ellos, lo que hace necesario definir un protocolo o conjunto

de reglas para su coordinación. En la práctica, esto suele implicar la asignación de una serie de capacidades especiales a un conjunto de nodos para garantizar el funcionamiento del sistema [10].

En el extremo del continuo de computación se encuentran los dispositivos IoT anteriormente introducidos, los cuales incluyen sensores, actuadores o *gateways* inteligentes conectados a la red. Estos dispositivos representan la fuente principal de datos en la mayoría de los sistemas, actuando como el primer nivel de interacción con el entorno. Realmente, el concepto *Internet of Things* (IoT) está relacionado con la conexión a internet de elementos de todo tipo, como pueden ser elementos cotidianos como los electrodomésticos o sensores que midan, por ejemplo, los niveles de contaminación en una ciudad, por lo que también es conocido como IoE (*Internet of Everything).* No obstante, la capacidad de computación de este tipo de dispositivos ha ido evolucionando paulatinamente hasta tener un mayor peso en las arquitecturas y sistemas de computación, incluso dando lugar a la definición de dos nuevos conceptos.

El IIoT (*Industrial Internet of Things*) engloba la aplicación del IoT en entornos industriales en los que su uso se focaliza en mejorar la observabilidad de los procesos de producción y logística con el objetivo final de optimizarlos. Esto se consigue mediante la sensorización de maquinaria para obtener datos en tiempo real, que posteriormente pueden ser analizados aplicando técnicas de Big Data o ejecutando algoritmos de IA. Por otra parte, el IoIT tiene en cuenta el aprovechamiento del incremento de la capacidad de computación de los dispositivos IoT para dotarlos con capacidades avanzadas de análisis, aprendizaje y toma de decisiones autónomas, incluyendo su inclusión en flujos más complejos de IA que engloben elementos de computación adicionales [12].

Finalmente, estos tres principales segmentos o espectros de la arquitectura de los sistemas de computación a nivel global convergen en el llamado continuo *Cloud-Edge-IoT*. Este concepto va más allá de la mera unión de todos los elementos (como recursos de computación, servicios desplegados o fuentes de datos) pertenecientes a todos los segmentos, puesto que trata de alcanzar una integración y colaboración fluida entre ellos [13]. Un caso de uso concreto consistiría en que los datos generados por los dispositivos IoT puedan ser procesados localmente por elementos del *edge computing* (para reducir la latencia, por ejemplo) y finalmente enviar los datos estrictamente necesarios a la nube. En esencia, el continuo actúa como un marco flexible y escalable que optimiza el uso de recursos computacionales y de red, adaptándose a las necesidades específicas de cada aplicación o escenario, sin necesidad de hacer una distinción estricta del segmento en el que se realiza realmente el procesamiento. Este cambio de paradigma se plasma perfectamente en los dos cambios de nombre consecutivos que ha realizado la asociación AIOTI (originalmente *Alliance for IoT Innovation* y actualmente *Alliance for AI, IoT and Edge Continuum*) en los últimos dos años, primero añadiendo el término *Edge Computing* para luego sustituirlo por *Edge Continuum* [14].

2.3. Adaptación de *cloud* a *edge*: *edge-native*

El concepto de *edge-native* fue introducido por primera vez por M. Satyanarayanan en 2019 [15]. Desde entonces, ha sido utilizado en múltiples foros por empresas como IBM, Cisco y otras, estableciéndose como el término más popular para referirse a la aplicación de principios de diseño *cloud-native* en dispositivos de computación en el *edge*. Estos principios fundamentales incluyen el diseño de sistemas en arquitecturas basadas en microservicios (logrando un despliegue modular de sistemas distribuidos), virtualización utilizando contenedores, gestión dinámica, orquestación y planificación, y uso de metodologías DevSecOps, entre otros [16]. Además, el lenguaje de programación Go, impulsado por Google, se ha convertido en el lenguaje más utilizado para el desarrollo de tecnologías *cloud-native,* entre las que se incluyen proyectos tan importantes como el mismo Kubernetes, su base de datos *etcd*, Helm o Prometheus.

Resulta evidente que existe un creciente interés en el mundo de la investigación sobre la transferencia de tecnologías y herramientas al *edge* para cumplir con dichos principios, aplicando características de agilidad, adaptabilidad y accesibilidad a los escenarios del *edge computing* [17]. Este interés se puede observar en el lanzamiento y la promoción de proyectos por parte de entidades relevantes cuya base de acción se enmarca en la promoción de desarrollos colaborativos con licencias código abierto, como la Linux Foundation (dentro de su línea de acción LF Edge [18]) o la Eclipse Foundation [19]. En este ámbito cobra especial relevancia la Cloud Native Computing Foundation (CNCF), un consorcio neutral que ha estandarizado y promovido principios y tecnologías nativas de la nube como Kubernetes, etcd, CoreDNS, gRPC, entre muchos otros. La CNCF se ha encargado de la organización y puesta en marcha de una serie de proyectos, que pueden estar en fase de consolidación o aún en fase de pruebas, bajo su propia tutela y específicamente orientados a solucionar desafíos propios del *edge computing* [20]. Además, entidades públicas como la Comisión Europea (EC) han promovido convocatorias de financiación relacionadas con esta temática [21], como el desarrollo de meta sistemas operativos (Meta-OS) para intentar gestionar la heterogeneidad de recursos presentes en el continuo de computación [22], o la orquestación inteligente distribuida de diversas cargas de trabajo en toda la infraestructura de computación de dicho continuo (IA) [23]. En resumen, la gran mayoría de estas iniciativas tienen como objetivo materializar el llamado continuo de computación *Cloud-Edge-IoT* (descrito en la sección anterior), cuyo éxito se basa en la aplicación de técnicas nativas de la computación en la nube a lo largo de todo el espectro computacional. Siguiendo el espíritu de código abierto promovido por Eclipse y la CNCF, la EC ha lanzado una ambiciosa iniciativa: EUCloudEdgeIoT [24], la cual pretende unir todas las acciones de investigación europeas que buscan desarrollar dichos continuos de computación con un conjunto común de tecnologías de código abierto, como se observará en la sección 3.4 de esta tesis doctoral.

Dentro del IoT Edge Working Group de la CNCF se llegó a un acuerdo sobre los principios de diseño que, a su juicio, deberían seguir las aplicaciones para ser consideradas como *edge-native*. Estos principios fueros descritos en un reporte lanzado en enero de 2023 [25] junto con una extensa comparativa con los correspondientes con *cloud-native* que se resume en la Tabla 2.1. El éxito de este reporte hizo que sus autores lanzaran una extensión en la que incluyeron una serie de principios prácticos para el diseño de aplicaciones *edge-native* [26].

Tabla 2.1: Comparativa entre los principios de las aplicaciones *cloud-native* y *edge-native*

Característica	Cloud-native	Edge-native
Patrón de diseño	Mayoritariamente microservicios con facilidad para ser escalados horizontalmente.	También suelen seguir el patrón de microservicios, pero teniendo en cuenta que su escalabilidad es limitada.
Modelo de datos	La persistencia de los datos se realiza en elementos centralizados a los que tienen acceso.	Necesitan técnicas alternativas para poder gestionar las pérdidas de conexión con los elementos de persistencia (p. ej. caché). A menudo manejan datos en *streaming* o tiempo real debido a su cercanía con la fuente de datos.
Elasticidad	Suelen actuar asumiendo que los recursos en los que se despliegan son ilimitados.	Su elasticidad es limitada debido a la limitación de recursos de computación, pero pueden escalar verticalmente moviendo su despliegue a un dispositivo cercano con mayor capacidad de computación.
Escala	Número reducido de instancias de servicios en pocas localizaciones.	Despliegue en un gran número de localizaciones y de dispositivos heterogéneos.
Orquestación	Despliegue de servicios de manera horizontal en nodos de características similares y agrupados.	Los servicios son desplegados de manera distribuida, frecuentemente basándose en su ubicación.
Gestión	Gestión totalmente centralizada que incluye mecanismos de automatización.	Las aplicaciones *edge-native* se gestionan utilizando una mezcla de métodos centralizados y remotos.

Red	Redes homogéneas, cableadas, confiables y con altas velocidades de conexión.	Redes heterogéneas, conexiones inestables y uso de una mayor variedad de protocolos de red y de medios de conexión.
Conocimiento del *hardware* subyacente	Prácticamente nulo al tratarse de máquinas con características uniformes.	Deben tenerlo en consideración debido a la naturaleza heterogénea de los dispositivos de la capa *edge*, por lo que requiere un mayor esfuerzo en su desarrollo.
Interacción con recursos externos	Raramente necesitan interactuar directamente con dispositivos externos.	Su interacción con elementos propios de IoT es muy frecuente al estar desplegados en localizaciones cercanas a este tipo de dispositivos.

2.3.1. *Edge computing* en el sector de los proveedores de servicios de *cloud computing*: soluciones comerciales actuales

En la actualidad, las empresas líderes del sector de la computación en la nube (AWS, Google, Microsoft...) han lidiado con el paradigma del *edge computing* como una extensión de su propio ecosistema de soluciones basado en la nube, dado que estos productos y sus tecnologías subyacentes están debidamente implementadas y testeadas por los miles de clientes que tienen en todo el mundo. Esto supone una transformación en la arquitectura, en la que los servicios que originalmente se ejecutarían en la infraestructura *cloud* de estos proveedores (es decir, en sus centros de datos) pasan a ejecutarse en las propias instalaciones de los clientes. En otras palabras, la nube es reemplazada por la infraestructura local del cliente para la ejecución de la parte de computación centralizada.

En esta misma línea, los proveedores de servicios han desarrollado sus propios dispositivos de *hardware* para el *edge*, diseñados para ejecutar sus soluciones propietarias de manera nativa y con una intervención mínima por parte del usuario final. Esta característica simplifica la adopción de la tecnología y facilita su integración en distintos entornos. Estos dispositivos son comúnmente conocidos como dispositivos *Plug-and-Play (PnP).*

Amazon Web Services (AWS) ofrece una solución denominada AWS IoT Greengrass para desplegar los servicios o módulos de procesamiento sobre nodos pertenecientes a la parte *edge* de la arquitectura, especialmente en dispositivos IoT que recopilan datos de los diferentes sensores que tienen conectados [27]. Esta solución permite realizar despliegues de tipo *serverless* en los dispositivos *edge* utilizando AWS Lambda, la solución *serverless* propia de AWS. Greengrass transforma los modelos que se crearon previamente en la nube en funciones Lambda, que luego se trasladan a la parte *edge* (donde pueden existir dispositivos capaces de ejecutar contenedores e incluso algoritmos de inferencia de IA) para ejecutarse. Para evitar problemas de pérdida de conectividad a la red, Greengrass crea un gemelo virtual del dispositivo en la nube cuando hay una conexión estable que asegura el cumplimiento del estado deseado; de lo contrario, el dispositivo se vuelve a sincronizar cuando se restablece la conexión y se calibra para asegurar nuevamente el estado deseado. Además, permite la comunicación entre dispositivos conectados a redes locales sin depender directamente de la parte *cloud*. AWS IoT Greengrass también ofrece la gestión de dispositivos desde una ubicación remota, permitiendo la incorporación y eliminación de módulos de *hardware* y *software* (incluso personalizados).

Otra solución de AWS para el edge es AWS Snowball Edge, que se trata de un tipo de dispositivo de la familia AWS Snowball diseñado para trabajar en las instalaciones del cliente [28]. Amazon ofrece tres versiones de este dispositivo, cada una optimizada para satisfacer distintas necesidades en cuanto a capacidad y rendimiento: (i) optimizado para almacenamiento y transferencia de datos (80TB de capacidad de almacenamiento utilizable), (ii) optimizado para almacenamiento con funcionalidad de computación EC2 (producto de AWS enfocado en proporcionar capacidades de computación bajo demanda) y (iii) optimizado para computación (equipado con 104 vCPUs y 416 GB de memoria). Al igual que los módulos de Greengrass, los dispositivos Snowball Edge pueden gestionarse localmente y a través de la nube. Asimismo, pueden ejecutar cargas de trabajo potentes para mover todos los datos almacenados localmente al servicio de almacenamiento AWS S3 en la nube y controlar los despliegues que se ejecutan en los dispositivos Greengrass. Finalmente, estos dispositivos pueden situarse en el nivel superior de la capa *edge* del continuo de computación (incluso ser considerados como *cloud on-premise*), ya que ofrecen grandes capacidades de procesamiento.

Microsoft Azure ofrece un catálogo de servicios completo para el *edge computing* dentro del marco Azure IoT Edge, que incluye la gestión remota de equipos, la virtualización de servicios en el *edge*, además de la asignación y control de cargas de trabajo remotas [29]. Esta tecnología está disponible bajo una licencia de código abierto MIT a través de la cuenta de GitHub de Azure, lo que permite a los desarrolladores desplegar e integrar las herramientas que ofrece Azure para el *edge* en su propia infraestructura. Sin embargo, la interoperabilidad no es completa, ya que su instalación e integración sigue dependiendo de los servicios en la nube ofrecidos por Azure.

De manera similar a Amazon, Microsoft ofrece una serie de dispositivos de computación potentes para trasladar los servicios de Azure a las instalaciones del cliente o a ubicaciones remotas (cerca de las fuentes de datos), reduciendo la cantidad de datos que se envían a la nube mediante el uso de las tecnologías de Azure Stack Edge. Esta oferta se divide en dos líneas de productos: Edge Pro Series, una línea enfocada a productos potentes para ubicarse tanto en un centro de datos local (Pro y Pro 2), como en equipos portátiles que pueden contener una fuente de alimentación ininterrumpida (Pro R); y Edge Mini Series, un dispositivo portátil limitado que cuenta con una batería [30].

Respecto a otro gigante tecnológico como es Google, Google Cloud integra soluciones *edge computing* dentro de su producto Google Distributed Cloud. Además, Google ideó una infraestructura completa para el continuo *Cloud-Edge* en el ámbito de las telecomunicaciones, ofreciendo servicios que pueden desplegarse en todos los niveles del espectro de computación (el extremo de la red de Google, el extremo de los operadores de telecomunicaciones, los centros de datos de los clientes y el *edge* remoto de los clientes), incluyendo la capa de red del proveedor de servicios de telecomunicaciones y la posibilidad de virtualizar los elementos de las redes 5G de los operadores de telecomunicaciones [31].

Como reflexión final, un aspecto común de todas las propuestas ofrecidas por las empresas dominadores del mercado del *cloud computing,* es el alto grado de dependencia del proveedor, lo que en última instancia se traduce en la falta de interoperabilidad entre ellas y otras tecnologías. Mientras que los mecanismos y tecnologías que se introducirán en secciones posteriores (distribuciones de K8s, KubeEdge y otros orquestadores) son abiertos y están orientados a cumplir con ciertos principios del diseño *cloud-native*, los sistemas comerciales presentados aquí crean ecosistemas cerrados, vinculados a interdependencias, configuraciones específicas, y patrones particulares de diseño de arquitecturas y de despliegue de aplicaciones, además de requerir un pago elevado para uso, entre otras limitaciones.

2.4.Intercambio de datos y protocolos de comunicaciones

2.4.1. Información de contexto

Los gestores de contexto o *context brokers* (CB) han jugado un papel crucial en las plataformas IoT, ya que actúan como elementos centrales que permiten persistir el valor más reciente de la información de contexto de dispositivos con una capacidad limitada, e incluso nula, de computación, y, en consecuencia, de almacenamiento. Esta información de contexto aporta información detallada sobre estos dispositivos IoT, pudiéndose clasificar esta información en (i) estática, que se actualiza un número limitado de veces o solamente en el momento de su creación o alta en el gestor de contexto (localización, responsable o modelo de un sensor o

actuador); y (ii) cambiante, que se actualiza periódicamente con una alta frecuencia (valor de la medición realizada por un sensor o el número de veces en las que un actuador se ha activado exitosamente) [32]. La importancia de los gestores de contexto en este tipo de plataformas se puede observar en el ejemplo concreto de FIWARE, que empezó siendo una plataforma de IoT en el momento de su creación en 2016 por la FIWARE Foundation, pero que ha ido ensanchado su espectro hasta acabar convirtiéndose en un conjunto de herramientas para desarrollar aplicaciones inteligentes. De hecho, el único requisito indispensable para que una solución sea considerada como *Powered by FIWARE* es la inclusión en su arquitectura del gestor de contexto de FIWARE: Orion [33].

Para lograr desarrollar la plataforma IoT FIWARE, una de las más destacadas con presencia en numerosas ciudades inteligentes, se creó a su vez una especificación para la representación y el intercambio de la información de contexto que manejan los gestores de contexto: NGSIv2 (New Generation Service Interfaces), permitiendo así que dispositivos IoT y servicios puedan intercambiar información de manera fluida utilizando un lenguaje común. En NGSI, esta información se modela en entidades utilizando el lenguaje JSON, que están compuestas por:

- Un **identificador único** (parámetro *id*)
- Un único **tipo de entidad** (parámetro *type*), que actúa al mismo tiempo de agrupador de entidades y como elemento descriptivo del tipo de atributos que se espera que presente la entidad.
- Una serie de **atributos** en las que se almacenan realmente la información de contexto. Estos atributos incluyen valores, marcas temporales (momento de creación y modificación del atributo) y metadatos (por ejemplo, la unidad del valor medido).

Además, el intercambio de datos en NGSIv2 está basado en API REST, por lo que este estándar define varios *endpoints* con distintos métodos HTTP para realizar diversas acciones. Por ejemplo, para obtener todas las entidades de un mismo tipo se ha de realizar una petición HTTP GET al *endpoint* "*/entities*", mientras que para dar de alta una nueva entidad se ha de enviar una petición POST a la misma ruta. Asimismo, NGSIv2 también ofrece mecanismos basados en el modelo de publicación y suscripción para informar en tiempo real a agentes externos en el momento en el que se producen cambios claves en la información de contexto, para permitir reaccionar a éstos consecuentemente [34].

No obstante, debido al aumento de la complejidad de las aplicaciones y necesidades de interoperabilidad en el entorno del IoT, se empezó a diseñar una evolución de este estándar para ampliar la información semántica de la información de contexto. El resultado fue el surgimiento de NGSI-LD, estandarizado por el ISG CIM (Industry Specification Group on cross-cutting Context Information Management) de la ETSI (European Telecommunications Standards Institute) [35], que está completamente basado en los principios de *Linked Data*, y por ello su lenguaje para la definición de las entidades pasa a ser JSON-LD [36]. Además, los

metadatos pasan a definirse como subatributos, reservándose algunos valores especiales como *unitCode,* para definir las unidades de los valores de los atributos, o *observedAt*, una marca temporal para especificar el momento en el que un valor fue medido. Este cambio permite establecer relaciones entre los atributos de varias entidades (mediante la introducción del nuevo tipo de atributos *Relationship*) junto con la integración con estándares semánticos como RDF (Resource Description Framework) [37], lo cual permite avanzar hacia un modelo más colaborativo. A este avance contribuyen de manera destacada la inclusión de mecanismos para la compartición de información de contexto entre varios gestores de contexto y por ende habilitando su uso en entornos distribuidos. Estos mecanismos se basan en la creación de diversos *Context Source Registrations* (CSRs o simplemente *registrations*), que se trata de un objeto de configuración específico de NGSI-LD a través del cual se informa a un CB sobre en qué otro CB puede encontrar cierta colección de entidades o incluso de sus atributos, ya que también pueden distribuirse atributos de una misma entidad en diversos CBs, por lo que en estas *registrations* es necesaria la definición de ciertos parámetros como la URL del CB externo o el tipo u ID de estas entidades. A nivel práctico, cuando se realiza una consulta a un gestor de contexto NGSI-LD para obtener una lista de entidades, éste comprueba primero las CSRs con el objetivo de encontrar coincidencias (por ejemplo, si se desea obtener entidades del tipo *Dominio*, el CB busca CSRs que apliquen a entidades de este tipo), para seguidamente realizar una petición distribuida al CB al que apunta cada una, cuyos resultados engloba en la respuesta final a la consulta original, siendo el proceso completamente transparente para el agente que realiza esta consulta [38].

No obstante, en el ámbito de los sistemas altamente distribuidos, estas operaciones presentan un desafío técnico elevado. Cuando el volumen de los resultados es elevado, el estándar NGSI-LD aplica un sistema de paginación, es decir, en cada petición solamente se devuelve un número predeterminado de resultados (por ejemplo, 20 entidades). Esta característica, aunque necesaria para garantizar la escalabilidad y eficiencia en las consultas de la información de contexto, introduce un problema cuando las consultas involucran atributos altamente dinámicos, puesto que, durante el proceso de paginación, podrían incorporarse nuevas entidades o modificarse los valores de los atributos de las ya consultadas, lo que impediría obtener una visión real y coherente sobre el contexto. Es por este motivo por el que es necesario establecer una colección consistente de entidades NGSI-LD resultantes para cada petición distribuida. Después de un proceso de deliberación en el ETSI ISG CIM, se propusieron dos soluciones:

- **Entity Maps:** esta solución se basa en congelar la colección de las entidades resultantes en una lista en la que solo se incluye sus ID.
- **Snapshots:** este enfoque congela la misma colección de entidades resultantes en una base de datos local.

Finalmente, decidieron adoptar la solución basada en Entity Maps con el propósito de reducir la cantidad de datos que intercambian los gestores de contexto,

incluyéndose en la versión 1.8.1 de la especificación de la API de NGSI-LD, aunque sin descartar el uso de Snapshots en el futuro. A nivel práctico, cuando se realiza una consulta de entidades a un CB, éste crea y almacena localmente un Entity Map estructurado como un objeto JSON en el que se mapean los ID de las entidades con vectores o listas de ID que identifican a las CSRs vinculadas con la entidad en cuestión, usando el valor especial *@none* cuando la entidad pertenece localmente al CB. Este Entity Map es consultado por el CB en futuras consultas o cuando la paginación de resultados es necesaria [39].

Tabla 2.2: Comparativa entre NGSIv2 y NGSI-LD

Característica	NGSIv2	NGSI-LD
Formato de datos	JSON	JSON-LD
Formato de los identificadores de las entidades	Cadenas de texto únicas	URIs
Relaciones entre entidades	Atributos simples	Establecimiento de relaciones (atributos de tipo *Relationship)* y uso de semántica
Arquitectura	Centralizada	Distribuida
Modelo de datos	Centrado en atributos	Enriquecido con gráficos RDF, uso del @context de JSON-LD

Actualmente existen varias implementaciones de gestores de contexto NGSI-LD, es decir, que implementen las funcionalidades y la API definida en su especificación, pero realmente destacan un par de ellas: Orion-LD de FIWARE [40] y Scorpio de NEC [41]. Scorpio ha sido desarrollado siguiendo un patrón de arquitectura basado en microservicios, que han sido desarrollados usando Java junto con Spring Framework, utilizando un bróker de mensajería Kafka como medio de comunicación común, mientras que Orion-LD se trata de una aplicación monolítica desarrollada completamente en C. De esta manera, Scorpio cuenta con una arquitectura más moderna que sigue un patrón de diseño *cloud-native*, mientras que se puede afirmar que, aunque no su arquitectura no sea *cloud-native*, Orion-LD aporta un mejor desempeño de su ejecución al ser C un lenguaje compilado que ofrece una mayor eficiencia para el tipo de operaciones que realiza un gestor de contexto.

Es necesario destacar que FIWARE lidera una iniciativa de colaboración llamada *Smart Data Models* para definir y compartir modelos de datos estandarizados y totalmente abiertos organizados en varios ámbitos de aplicación como ciudades inteligentes, energía, salud o logística [42]. Cada modelo de datos se

define en un repositorio en GitHub que cuenta con un esquema en el que se describen los tipos de datos y su estructura, una especificación en la que se describe el modelo de datos y varios ejemplos en formatos NGSIv2 y NGSI-LD. El grupo de investigación SATRD, del cual forma parte el doctorando, ha contribuido activamente a esta iniciativa con la creación, mejora y uso de diferentes modelos de datos en diversos proyectos de investigación, destacando la creación de un modelo de datos para el transporte marítimo [43].

Finalmente, aunque esta información de contexto ha estado tradicionalmente relacionada con el mundo IoT, recientemente se ha tenido en cuenta su potencial para la monitorización de todo tipo de elementos de un ecosistema tecnológico, especialmente en un entorno heterogéneo como el continuo *Cloud-Edge-IoT*, ya que los diferentes elementos de computación que lo construyen o incluso los servicios desplegados en ellos pueden modelarse utilizando un modelo de datos adecuado [44][45]. Además, los mecanismos de operaciones distribuidas anteriormente descritos permiten que encajen en el paradigma de este continuo de computación donde los elementos centralizados tienden a reducirse, como se ha explicado en los primeros compases del estado del arte. Es por estos motivos por los que los gestores de contexto van a ser una parte fundamental de esta tesis doctoral.

2.4.2. Comunicaciones síncronas y asíncronas

En un ecosistema en el que predominan los sistemas que siguen una arquitectura distribuida como es el continuo *Cloud-Edge-IoT*, la elección entre comunicaciones síncronas y asíncronas resulta ser crucial, de ahí que la balanza deba decantarse según los requisitos específicos de las aplicaciones. Estos requisitos pueden ir desde necesidades de latencia o capacidad de respuesta en tiempo real hasta la tolerancia a fallos. Las comunicaciones síncronas, caracterizadas por su modelo de petición-respuesta con una gran dependencia temporal entre emisor y receptor, son ampliamente utilizadas en aplicaciones que requieren respuestas inmediatas, como lo son las interacciones cliente-servidor en las APIs REST. Por otro lado, las comunicaciones asíncronas permiten un desacoplamiento temporal, habilitando la emisión de mensajes sin necesidad de esperar confirmaciones inmediatas, ya que suelen desencadenar procesos en segundo plano que requieren un cierto tiempo de computación [46]. Esta característica las hace ideales para arquitecturas escalables y basadas en eventos (*event-driven architectures* o EDAs) que frecuentemente hacen uso de *brokers* de mensajería como Kafka o NATS. A modo de resumen, en la siguiente tabla se presenta una comparación detallada de ambos enfoques tratando una serie de aspectos.

Tabla 2.3: Comparativa entre comunicaciones síncronas y asíncronas

Aspecto	Comunicaciones síncronas	Comunicaciones asíncronas
Definición	El emisor espera a la respuesta del receptor a su petición antes de continuar con el proceso.	El emisor envía un mensaje y continúa el proceso sin esperar a la respuesta. Proceso no bloqueante.
Dependencia	El emisor y el receptor están altamente acoplados ya que tienen que estar disponibles al mismo tiempo.	Bajo acoplamiento dado que el emisor y el receptor pueden operar independientemente.
Comportamiento	Comportamiento bloqueante puesto que el emisor bloquea la ejecución del proceso hasta que recibe una respuesta del emisor.	Comportamiento no bloqueante ya que el emisor continúa con la ejecución del proceso sin tener en cuenta la posible respuesta del receptor.
Escalabilidad	Escalabilidad limitada debido al comportamiento bloqueante.	Altamente escalable gracias al procesamiento paralelo.
Casos de uso	Consultas a bases de datos, REST APIs, sistemas con necesidad de respuesta.	Sistemas basados en eventos, transmisión de datos en tiempo real, colas de mensajes.

Las comunicaciones síncronas se plasman en las ampliamente utilizadas y establecidas en la industria APIs REST, en las que un cliente realiza peticiones HTTP a los debidos *endpoints* de un servidor utilizando una serie de métodos predefinidos con un significado asociado (GET, POST, PUT, PATCH y DELETE). Asimismo, cuentan con una gran cantidad de librerías para escribir tanto clientes como servidores en todos los lenguajes de programación, programas cliente (*curl*, Postman, Bruno...) y una herramienta de documentación prácticamente estandarizada como es OpenAPI. En el ámbito de los microservicios, su caso de uso más frecuente es en procesos en los que se quiere realizar acciones de manera inmediata, como obtener una serie de datos de una base de datos, dar de alta un elemento en un sistema o eliminarlo.

Aunque las REST APIs representan un enfoque ampliamente adoptado para las comunicaciones síncronas en arquitecturas distribuidas, las crecientes demandas de baja latencia, eficiencia en el uso del ancho de banda y soporte para *streaming* bidireccional han impulsado la evolución hacia alternativas más flexibles. En este

contexto, gRPC [47] surge como una solución moderna que, si bien mantiene ciertas similitudes con las REST APIs al basarse en HTTP/2, introduce capacidades asíncronas que la posicionan como una opción más adecuada para entornos distribuidos y de alto rendimiento [48]. La citada mejora en la eficiencia se consigue en gran parte gracias a que gRPC utiliza *Protocol Buffers* (Protobuf) para serializar binariamente los mensajes, puesto que reduce el tamaño de los datos transmitidos en comparación con formatos basados en texto como JSON [49], que podría prácticamente considerarse como el estándar en las API REST.

En cuanto a las comunicaciones asíncronas, este enfoque se basa principalmente en el uso de *brokers* de mensajería, que actúan como intermediarios para gestionar el intercambio eficiente de mensajes entre aplicaciones adoptando un modelo de publicación-suscripción en contraposición al modelo de petición-respuesta. Además, este modelo permite que un único mensaje sea recibido por varios receptores simultáneamente (relación 1:N). Debido al incremento y consolidación de su uso en sistemas distribuidos, se hizo patente la necesidad de crear una herramienta de documentación similar a OpenAPI en la que se pudieran describir los canales de comunicaciones entre las aplicaciones y los *brokers* de mensajería, incluyendo principalmente información sobre el tipo de operación (envío o recepción) y formato de los mensajes intercambiados. Por este motivo, surgió la herramienta de código abierto llamada AsyncAPI, cuyas características son similares a OpenAPI, como queda reflejado en el uso de YAML para describir las comunicaciones adaptando el formato original de OpenAPI (incluso este formato es compatible con AsyncAPI) o en la inclusión de herramientas de generación de código cliente [50]. Del mismo modo, la CNCF ha adoptado una iniciativa para estandarizar la descripción de eventos en el ámbito de las EDAs y ser integrada en los *brokers* de mensajería: CloudEvents, que además aporta integración nativa con algunos *brokers* de mensajería como Kafka [51] o NATS.

Apache Kafka ha sido uno de los *brokers* de mensajería más utilizados desde su lanzamiento, estableciéndose como un referente en la industria especialmente para casos de uso en los que se necesita manejar eficientemente una transmisión de datos a gran escala [52]. Esta solución, desarrollada en Java, está basada en una arquitectura distribuida basada en *logs*. Esto implica que Kafka organiza y almacena los datos como una secuencia inmutable de registros conocidos como *logs*, que solamente pueden ser añadidos al sistema, sin poder ser modificados ni eliminados. Además, estos registros pueden ser distribuidos entre diferentes *brokers* de Kafka para aumentar la escalabilidad y tolerancia a fallos del sistema. En un principio hacía uso de Apache ZooKeeper para la coordinación de los diferentes *brokers* pertenecientes a un clúster y la gestión de sus metadatos, pero la dependencia de esta tecnología se eliminó al incorporar de manera nativa Kraft (Kafka Raft Metadata), ya que ZooKeeper presentaba problemas de escalabilidad [53].

Se podría afirmar que el principal inconveniente de Kafka es su gran consumo de recursos y la complejidad para realizar un despliegue en producción de manera

eficiente, en gran parte debido a su uso de la JVM al estar desarrollado en Java, aunque se han realizado esfuerzos para mejorar este aspecto como la anteriormente citada sustitución de ZooKeeper. Por este motivo, Redpanda se desarrolló en C++ y utilizando patrones de diseño *cloud-native* como una alternativa de fácil reemplazo a Kafka, ya que es totalmente compatible con su API. La intención de sus desarrolladores era tanto reducir su tamaño como simplificar su despliegue y configuración, para así mejorar su desempeño [54].

Finalmente, el *broker* de mensajería o "middleware orientado a mensajes" NATS se desarrolló para ser utilizado como medio de comunicación común entre los componentes que componen un sistema distribuido moderno, teniendo en cuenta especialmente las características de los entornos híbridos *edge-cloud* en su diseño. Es por este motivo que no intenta competir con Kafka o Redpanda en cuanto al procesamiento masivo de datos centralizados, sino que prioriza la capacidad de ser desplegado en gran variedad de entornos y asegurar la entrega y recepción de mensajes en arquitecturas complejas pertenecientes al continuo de computación [55].

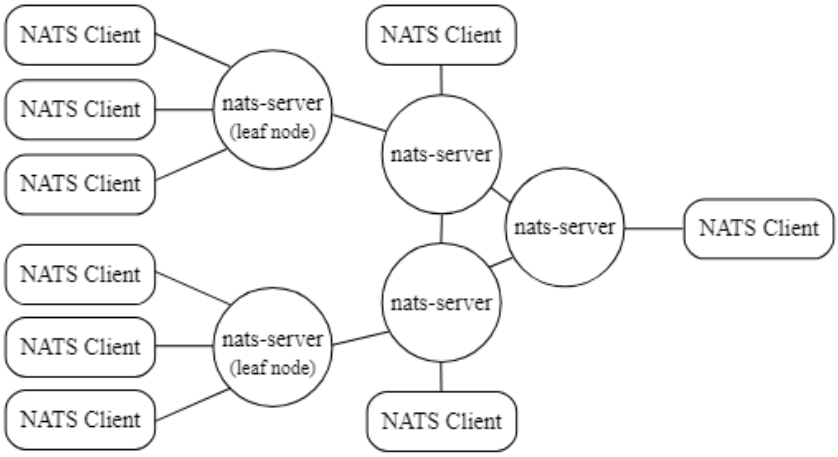


Figura 2.2: Federación de diferentes *brokers* NATS a lo largo del continuo [57]

Por último, la siguiente tabla ilustra las principales diferencias entre NATS y Redpanda, aportando información más detallada sobre ambas tecnologías [56].

Tabla 2.4: Comparativa entre los *brokers* NATS y Redpanda

Característica	NATS	Redpanda
Arquitectura	Sistema ligero de mensajería cuyo diseño se centra en la simplicidad, ligereza y rapidez.	*Broker* de mensajería enfocado a la gestión de un flujo de grandes cantidades de datos de manera centralizada. Diseñado como alternativa a Kafka.
Desempeño	Optimizado para aplicaciones con necesidades de baja	Optimizado para la recepción y tratamiento de grandes

	latencia o de comunicaciones en tiempo real.	cantidades de datos, por lo que prioriza la potencia. Mejora el rendimiento de Kafka en casos con un alto monto de trabajo.
Requisitos de despliegue	Empaquetado como un binario estático de pequeño tamaño que puede desplegarse incluso en dispositivos con capacidad de computación limitada como SBCs. Sus requerimientos aumentan ligeramente al activar JetStream. Asimismo, soporta una gran variedad de arquitecturas de CPU (x86_64, ARM64, ARMv7...) y OS (Linux, Windows, MacOS y FreeBSD).	Requerimientos de *hardware* más elevados pensados para la computación en la nube: cada *broker* debe desplegarse en una máquina diferente que tenga al menos 2 núcleos de CPU físicos (aunque se recomiendan 4) y 2 GB de RAM por cada uno de estos núcleos. Solamente soporta arquitecturas de CPU x86_64 y Linux como OS.
Garantía de entrega de mensajes	Al menos una vez, como mucho una vez y exactamente una vez (activando JetStream).	Al menos una vez, exactamente una vez.
Retención de mensajes y persistencia	No incorpora persistencia de mensajes por defecto, pero puede soportarla si se activa su extensión JetStream.	Incorpora persistencia por defecto.
Mecanismos de federación o agrupamiento	Implementa mecanismos de federación entre instancias de *brokers* en entornos distribuidos pensado para ser utilizados en entornos *edge-cloud*. Los *brokers* pueden agruparse dinámicamente en clústeres, y estos formar grupos de clústeres llamados *superclusters*. Además, incorpora *leaf nodes* que actúan como intermediarios entre un clúster de *brokers* y clientes remotos, para de esta manera permitir conectar a estos clientes al clúster como si se tratara de un único	No incluye mecanismos de federación entre *brokers*, ya que no es su propósito. No obstante, implementa mecanismos de replicación de datos entre *brokers* o despliegues totalmente sincronizados.

	cliente, o incluso crear copias de flujos locales en el clúster. Dentro de estos sistemas complejos, los mensajes se enrutan selectivamente dependiendo de las suscripciones [57].	
Casos de uso	Optimizado para el *edge computing* debido a la distribución de mensajes con baja latencia, diseño ligero y simple. Estas características lo convierten en un *broker* ideal para arquitecturas de microservicios distribuidos en elementos de computación heterogéneos en el continuo.	Diseñado para casos de uso en los que se tienen que procesar *streams* de datos con alta tasa de envío en arquitecturas más centralizadas en lugar de distribuidas. Claros ejemplos serían: sistemas de agregación y clasificación de mensajes o análisis de datos en sistemas basados en técnicas de Big Data.

2.5. Arquitecturas de red en el *edge computing*

Tal y como se ha ido apuntando en las anteriores secciones en las que se han descrito las características de los entornos asociados con el *edge computing*, a diferencia de lo que ocurre en los mismos del *cloud computing*, éstos presentan una mayor heterogeneidad en cuanto a su topología, siendo las más comunes topologías dinámicas en las que los dispositivos se conectan y desconectan continuamente, en cuanto a protocolos (5G, LoRaWAN, ZigBee...) e incluso respecto a su medio físico de conexión, ya que el uso de redes inalámbricas aumenta significativamente en este entorno, introduciendo a su vez una mayor inestabilidad en las conexiones [58]. Asimismo, otro desafío con el que tienen que lidiar estas redes es la gestión de la conexión de un gran número de dispositivos, ya que esta cifra aumenta a medida que se desciende en los bloques de arquitectura del *Cloud-Edge-IoT* como se observa en la Figura 1.1, entrando en acción dispositivos IIoT e IoIT [59].

En lo que respecta a la capa de red del modelo OSI (Open Systems Interconnection, norma ISO/IEC 7498-1), normalmente los dispositivos están conectados a redes internas y privadas, sin IPs públicas a nivel de internet o directamente en su mismo dominio, haciendo que en ciertos casos su enrutamiento o conexión directa, al igual que la exposición pública de servicios desplegados en estos entornos, sea un desafío por sí mismo [58]. Además, cuando entran en juego tantas redes diferentes, los dispositivos de seguridad como los cortafuegos añaden una mayor complejidad. Es por este motivo que se podría afirmar que el verdadero

reto consiste en encontrar una solución que proporcione conectividad entre los servicios desplegados en los diferentes elementos de computación, de manera que abstraiga de estas conexiones a bajo nivel al usuario final, pero siempre sin olvidar las características subyacentes de cada elemento de computación. Como primer paso hacia el enfoque de esta problemática, es necesaria una investigación sobre las diferentes soluciones existentes para evaluar las potenciales candidatas sobre las que se podría construir una solución o formar parte de ella.

2.5.1. Soluciones basadas en la capa de red

El establecimiento de túneles para la interconexión de diferentes subredes es una práctica habitual para dotar de visibilidad a equipos situados en diferentes áreas geográficas o administrativas, además de proporcionar una capa adicional de seguridad dado que el tráfico se envía de manera encriptada a través de ellos. Las VPN tradicionales están basadas en el patrón de arquitectura cliente-servidor, en el que un nodo cliente establece en primer lugar una conexión con el servidor VPN con la debida autenticación (usualmente indicando un usuario y una contraseña), para seguidamente redirigir todo su tráfico (o una parte de él aplicando un filtrado de destino previo, como un rango de IPs) encriptado hacia este servidor, el cual se encarga de desencriptarlo para redirigirlo a su destino final. Existen una gran variedad de protocolos de VPN (IPSec, OpenVPN, WireGuard...) para el establecimiento de túneles punto a punto, así como de implementaciones de esta tecnología, destacando tradicionalmente OpenVPN como solución de código abierto o aplicaciones propietarias como NordVPN o AnyConnect de CISCO, que incluso ofrecen VPNs como servicio.

La plataforma ExpressVPN ha desarrollado recientemente un protocolo propietario (aunque algunos de sus módulos han sido publicados con licencias de código abierto) llamado Lightway Core, el cual intenta solucionar los problemas de las VPN tradicionales ofreciendo conexiones utilizando UDP y TCP según el caso de uso [60]. Además, mejora la velocidad de sus conexiones y realmente ofrece una estabilidad mayor en comparación a sus alternativas. No obstante, esta no es la solución más interesante o innovadora para el ámbito de red del *edge computing*, ya que además de ser propietaria, se limita a mejorar las VPN tradicionales. Es en este ámbito donde emerge WireGuard, que ha logrado recientemente establecerse como una alternativa a las VPN tradicionales, tanto a nivel de protocolo de cifrado y de establecimiento de túneles, como de autenticación de clientes, e incluso del diseño de su arquitectura, yendo un paso más allá de las mencionadas arquitecturas cliente-servidor. Asimismo, es necesario destacar un aspecto de WireGuard que lo sitúa como una tecnología de interés en esta tesis doctoral: esta tecnología que ha revolucionado un mecanismo tan popular como las VPNs nació como resultado de un proceso académico, como se puede comprobar en el artículo de investigación realizado por su creador, Jason A. Donenfeld, en el que se detalla el diseño e

implementación del protocolo de WireGuard [61]. En cuanto a su funcionamiento, WireGuard encapsula los paquetes IP sobre UDP y emplea el llamado *Cryptokey Routing*, en el cual se asocian claves públicas a una lista de direcciones IPs de túneles permitidas dentro del mismo túnel, por lo que cada interfaz de red cuenta con una clave privada y un listado de *peers* con su clave pública para poder autenticarse entre ellos. De este modo, un *peer* puede actuar tanto como "cliente" como "servidor" dependiendo del túnel.

La seguridad y ligereza de esta tecnología (menos de 4000 líneas de código escritas en C y bajo consumo de recursos) junto con su compatibilidad con múltiples arquitecturas, sistemas operativos y la existencia de librerías para diversos lenguajes, convierten a WireGuard en una opción ideal para el establecimiento de túneles de red en entornos del *edge computing*, ya sea con el objetivo de dotar a nodos superiores acceso a dispositivos restringidos, añadir a estos a la red global del sistema, o incluso la encriptación del tráfico en entornos de *zero trust* (ZT), como se demuestra en trabajos existentes [62]. El autor de esta tesis doctoral puede respaldar con su propia experiencia estas afirmaciones, ya que esta tecnología fue elegida para el establecimiento y configuración de las VPNs necesarias en la infraestructura del grupo de investigación SATRD después del descubrimiento de sus grandes beneficios en gran parte gracias a la investigación realizada en esta tesis. Su facilidad de configuración e integración con productos de gestión de código abierto como OPNsense la hacen una elección perfecta para todo tipo de entornos profesionales. Como casos de uso de relevancia en entornos *cloud-native*, es interesante mencionar que algunas implementaciones del CNI (Container Network Interface, una especificación definida por la CNCF para configurar las interfaces de red de los contenedores [63][64]), comúnmente denominadas simplemente CNIs, como Cilium, incorporan WireGuard de manera nativa para reforzar la seguridad encriptando el tráfico entre *Pods* o incluso nodos de Kubernetes [65], así como otros proyectos de la CNCF como *kube-vip* usan WireGuard para permitir la exposición de servicios de K8s a través de una red privada y distribuida a la que pueden conectarse diferentes nodos pertenecientes a diferentes clústeres para usar dichos servicios [66].

La técnica llamada NAT (Network Address Translation) se diseñó en su día para intentar reducir el uso de IPs públicas, ya que el rango de direcciones IPv4 disponibles es limitado (4.294.967.296 de direcciones únicas), por lo que el aumento exponencial de dispositivos conectados puso en serio peligro la disponibilidad de IPs en todo el mundo. La idea detrás de NAT es sencilla: en una red local, se asigna direcciones IP privadas a los dispositivos conectados, mientras que a su vez se les asigna una única IP pública para su conexión con el exterior, permitiendo un gran ahorro de direcciones públicas. El router que habilita la conexión de esta subred con el exterior, se encarga de las traducciones dinámicas (IP pública a la correspondiente IP privada), así como de la redirección de ciertos puertos públicos a los dispositivos internos correspondientes (conocido como *port forwarding*) [67]. Sin embargo, esta técnica presenta ciertas limitaciones en redes dinámicas y

distribuidas como las propias del *edge computing*, además de limitada compatibilidad con las comunicaciones P2P o en tiempo real (VoIP, videollamadas...). Es por este motivo por el que se empezaron a diseñar técnicas de NAT transversal, que se podría definir como un conjunto de técnicas y estrategias diseñadas para permitir que dispositivos conectados a diferentes redes internas puedan conectarse directamente entre sí [68]. Aunque NAT es un mecanismo puramente de la capa 3, NAT transversal no se circunscribe solamente a esta capa del modelo OSI, sino que también abarca la capa 4 en algunas soluciones que dependen de protocolos de transporte e involucran la gestión de puertos, e incluso capas más superiores en casos específicos en los que se necesita negociar conexiones en la capa de aplicación.

Para revolver la problemática anteriormente descrita, NAT transversal utiliza varios métodos, como el descubrimiento de puertos, el uso de servidores intermedios para el descubrimiento de IPs públicas, y técnicas específicas como el *hole punching*, que se trata de una técnica que permite establecer conexiones directas entre dispositivos conectados a redes privadas detrás de NAT al aprovechar las rutas creadas (en la tabla de estado interna del router) por conexiones salientes desde la misma red privada [68]. Algunos de los protocolos de NAT transversal más utilizados son los siguientes:

- **STUN** (Session Traversal Utilities for NAT): este protocolo con arquitectura cliente-servidor y definido en el RFC 8489 de la IETF, permite a los dispositivos descubrir su IP pública, el tipo de NAT que utiliza su red privada y el puerto externo asociado con su puerto local.
- **TURN** (Traversal Using Relays around NAT): este protocolo (RFC 8155) se diseñó como una extensión de STUN para solucionar el caso en el que la conexión directa entre dispositivos internos no es posible. TURN establece un servidor intermedio (comúnmente conocido como *relay*) para habilitar su conexión, aunque como consecuencia añade una mayor latencia.
- **ICE** (Interactive Connectivity Establishment): definida en el RFC 8445 de la IETF, esta técnica combina las dos anteriores (STUN y TURN) para intentar encontrar la conexión más directa posible, recurriendo a nodos intermedios solo cuando es estrictamente necesario.
- **UPnP** (Universal Plug and Play): es un conjunto de protocolos de red diseñados para facilitar la conectividad automática y la interoperabilidad entre dispositivos dentro de una misma LAN, por lo que, a diferencia de los anteriores, no necesita un servidor exterior. Por ejemplo, un dispositivo puede pedir al router la apertura de puertos para establecer conexiones elementos externos. Sin embargo, esta automatización puede presentar grandes riesgos de seguridad, por lo que está normalmente deshabilitado en redes corporativas.

Finalmente, es importante aclarar que el uso de ambas tecnologías no es incompatible, sino que se pueden combinar para dar con soluciones más eficientes. Es en este ámbito donde destaca Tailscale, una herramienta empresarial que permite conectar dispositivos y servicios que están conectados a diferentes redes. Tailscale utiliza WireGuard como tecnología para crear túneles entre nodos, pero estas conexiones son gestionadas por un elemento central llamado Coordination Server, que se encarga de dotar de una capa adicional de inteligencia, configuración y automatización a este proceso. Además, utiliza técnicas de NAT transversal para conectar los nodos ya que, en entornos reales, estos se encuentran frecuentemente detrás de redes locales que utilizan NAT, por lo que las conexiones de WireGuard no se pueden establecer directamente. Sin embargo, Tailscale ha ido un paso más allá desarrollado su propio protocolo de red llamado DERP (Designated Encrypted Relay for Packets) para el uso de TURN en los servidores que actúan como *relay*, ya que este protocolo presenta limitaciones para sus necesidades. La principal innovación es que DERP utiliza HTTPS junto con las claves de WireGuard [69].

2.5.2. Soluciones basadas en la capa de aplicación

Las soluciones basadas en la capa de aplicación ofrecen un enfoque más flexible y dinámico para resolver los desafíos de conectividad propios del *edge computing* descritos en el subapartado previo, especialmente en entornos distribuidos y heterogéneos, ya que, a diferencia de las soluciones basadas en la capa de red, las cuales se centran en el transporte de paquetes, las soluciones de capa de aplicación abstraen los detalles subyacentes de la red, además de proporcionar mecanismos para la gestión de la comunicación, la seguridad y la topología directamente en la capa de aplicación.

En este contexto, *libp2p* destaca como una herramienta clave para la creación de redes descentralizadas, capaces de adaptarse fácilmente a las características de diferentes entornos de red. A diferencia de lo que ocurre en las clásicas comunicaciones cliente-servidor, donde los servidores centrales actúan como nodos de control, las redes P2P no requieren de estos elementos centralizados, puesto que los nodos se comunican directamente entre ellos ejerciendo un tratamiento entre iguales, sin jerarquías ni distinciones. La definición que se puede encontrar en su documentación oficial es clara y concisa: "*libp2p* es un *framework* para el establecimiento de redes P2P que habilita el desarrollo de aplicaciones basadas en comunicaciones P2P. Este *framework* está formado por una colección de protocolos, especificaciones y librerías que facilitan las comunicaciones P2P entre los participantes de la red, conocidos como *peers*" [70][71]. Para añadir un poco de contexto, esta tecnología nació como una primera implementación de la capa de conexión de IPFS (InterPlanetary File System), que se trata de un sistema de intercambio de archivos totalmente descentralizado mediante direccionamiento basado en los contenidos, pero debido a su alto potencial y al incremento de su uso

en otros proyectos, se decidió continuar su desarrollo como un proyecto independiente.

En la red P2P de *libp2p* cada nodo o *peer* cuenta con un par de claves (pública y privada) para establecer canales de comunicaciones seguras entre ellos. Esta clave privada se utiliza a su vez para la creación de un identificador único para cada *peer* (*PeerID*), ya que éste es el resultado de realizar un *hash* criptográfico a su clave pública, que al codificarse en base58 se puede representar como una cadena de texto (p. ej. *QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N*). Debido a la flexibilidad de este tipo de redes, es necesario definir sistemas de direccionamiento con esta capacidad. Por este motivo, las *multiaddress* se diseñaron para codificar múltiples capas de información de direccionamiento en una sola entidad, siendo a su vez fácilmente legibles por humanos y optimizadas a nivel de máquina. A nivel práctico, en estas *multiaddress* se combinan varios protocolos (incluso los propios de *libp2p* como *Circuit Relay* usando el nombre *p2p-circuit*) junto con direcciones y puertos de red. Por ejemplo, la dirección */ip4/198.51.100.0/tcp/4242/p2p/QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N* significa que un *peer* con un ID (*Qmy...)*, es accesible vía la dirección IP pública 192.51.100.0 y el puerto TCP 4242.

En las redes P2P el proceso de descubrimiento de otros nodos y el enrutamiento hacia ellos es de vital importancia, ya que cada nodo debe descubrir y comunicarse con otros nodos sin utilizar un servidor central. Para este propósito se utilizan unos nodos de entrada (*boot nodes*), que deben de ser manualmente seleccionados y configurados. En *libp2p* ambos procesos ocurren simultáneamente mediante unos métodos definidos, siendo mDNS (*multicast* DNS) el más simple de ellos, en el que un *peer* envía una petición de tipo *broadcast* (se distribuye a todos los nodos) a su red local, que es respondida por los demás nodos con su propia *multiaddress*. En cuanto a las redes no locales (WAN), el método más utilizado para este proceso no es nada nuevo, ya que está basado en Kademlia Distributed Hash Table (Kad-DHT), que fue presentado por Petar Maymounkov y David Mazières en el artículo de investigación titulado *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric* [72] en 2002, pero que aún sigue vigente en el ámbito de P2P debido a su robustez y eficiencia, ya que permite reducir el número de saltos necesarios para encontrar un nodo o información sobre este, a la vez que manteniendo una alta escalabilidad incluso con un elevado número de nodos pertenecientes a la red. Como se trata de una tabla de tipo clave-valor distribuida en el que sus registros están esparcidos por la red, los *peers* han de preguntar a los demás *peers* hasta encontrar el registro deseado. Sin embargo, esta búsqueda no se realiza de manera aleatoria, sino que se basa en una métrica de distancia que representa la distancia entre dos *PeerIDs,* de manera que permite encontrar un número previamente definido de nodos más cercanos. Dada la complejidad de este proceso, solo se ha realizado una pequeña introducción a este, ya que está ampliamente descrito tanto en el artículo anteriormente mencionado como en la documentación de libp2p [73][74].

En cuanto a las comunicaciones entre nodos, *libp2p* emplea una abstracción de *streams* en lugar de depender de modelos basados en conexiones tradicionales, como TCP. Esta abstracción permite establecer canales de comunicación bidireccionales, en los que ambos nodos pueden enviar y recibir datos simultáneamente, mejorando así la eficiencia en las comunicaciones. Además, estos *streams* pueden operar sobre cualquier protocolo de la capa de transporte como TCP, WebSockets o QUIC. Dado que es común que los *peers* ejecuten múltiples aplicaciones al mismo tiempo, *libp2p* implementa una técnica denominada *stream multiplexing*, la cual permite la creación de varias conexiones virtuales sobre una misma conexión, de manera que los *peers* pueden crear diferentes *streams* de mensajes para cada aplicación diferente. Esto supone un aumento considerable de la eficiencia en las comunicaciones entre nodos, ya que, si esto no fuera posible y se utilizara el mismo *stream* para todas las aplicaciones, se provocaría un cuello de botella debido a que su uso simultáneo no sería posible. Además, *libp2p* también ofrece un mecanismo de comunicación basado en el modelo de publicación y suscripción, totalmente distribuido.

Como se ha descrito anteriormente, es esperable que la mayoría de los *peers* en una red P2P se encuentren en redes privadas que utilizan NAT y cortafuegos, por lo que *libp2p* también implementa mecanismos propios de NAT transversal. Esta librería se diseñó para intentar evitar la centralización que a menudo necesitan los protocolos anteriormente descritos STUN e ICE, de manera que cualquier elemento de la red puede convertirse en el elemento que se encarga de la coordinación del proceso de descubrimiento de los demás *peers*, ya que no es necesario ningún conocimiento previo de los nodos pertenecientes a la red y, además, los protocolos utilizados para este propósito han sido diseñados para consumir recursos mínimos. No obstante, en entornos reales a menudo se necesita que al menos una parte reducida de los *peers* tenga una IP pública o que sea directamente accesible.

El protocolo *Identify* ofrece una funcionalidad similar a STUN, pero adaptado a las redes P2P, ya que permite a un nodo descubrir su IP pública (o la lista de estas) y puerto conectándose a otro nodo, además de inferir si está detrás de NAT. Como este protocolo no permite averiguar si el nodo en cuestión puede ser directamente accesible por su IP pública determinada por Identify, es necesario un mecanismo adicional, así que esta información puede obtenerse mediante el uso del protocolo *AutoNAT*. Este protocolo consiste en que un nodo pida a otro nodo que intente realizar conexiones a una colección de direcciones del primero y así determinar su accesibilidad dependiendo del resultado del establecimiento de estas conexiones. De esta manera, si el nodo no puede ser directamente contactado con ninguna de sus IP públicas, se determinará que se encuentra detrás de NAT y su dirección pública pertenece a su *router* de salida.

El método más sencillo para establecer comunicaciones entre dos *peers* en los que al menos uno de ellos esté situado en una red privada, o que no compartan el mismo protocolo de transporte, es el uso de *Circuit Relay* [75], que a su vez está

inspirado por TURN, en el que el nodo *relay* actúa como intermediario (o incluso como un *proxy*) enrutando el tráfico entre ambos *peers*. Este tipo de comunicaciones están encriptadas entre ambos extremos, por lo que el nodo intermedio no puede leer o modificar su contenido. En primer lugar, los nodos privados deben realizar una petición de reserva al nodo *relay*, que és público, para que escuche las conexiones entrantes en su nombre. Los nodos *relay* pueden descubrirse dinámicamente vía el protocolo *AutoRelay*, que hace servir los métodos de descubrimiento y enrutamiento basados en Kad-DHT que fueron descritos antes. Después, si otro nodo quiere establecer una conexión con este nodo privado, debe realizar una petición al nodo *relay* para que transmita la conexión al nodo privado, para finalmente establecer una conexión entre los dos nodos en lo que toda la información pasará por el nodo *relay.* Sin embargo, aunque este método permite superar los posibles obstáculos en la conexión de nodos privados y frecuentemente es la única solución posible, su eficiencia es limitada ya que los nodos *relay* añaden latencia a las comunicaciones y se acaban convirtiendo en elementos centralizados que soportan una gran carga de tráfico y de responsabilidad, además de en un posible punto único de fallo en ciertas arquitecturas de red.

Un método más complejo para solucionar este problema sería el uso de los nodos *relay* (junto con el *Circuit Relay* previamente establecido) solamente para establecer comunicaciones directas entre nodos, en lugar de actuar como intermediarios en todas las comunicaciones. Este método fue explorado por los desarrolladores de *libp2p*, así que después de realizar una implementación propia y posteriormente varias pruebas para comprobar su validez decidieron describir su funcionamiento y resultados obtenidos en un artículo [76]. En primer lugar, ambos nodos han de comunicarse con el nodo *relay* para lograr establecer un *Circuit Relay*, tal y como se ha descrito en el párrafo anterior. Después, ambos nodos utilizan el protocolo DCUtR (Direct Connection Upgrade through Relay) para coordinar el establecimiento de una conexión directa tomando como punto de partida el *Circuit Relay* previamente establecido. Este protocolo ha sido diseñado para ser muy ligero, reduciendo el intercambio de datos a solo 500 bytes en cada sentido de la conexión. El primer paso en DCUtR es el envío por parte de un *peer A* de un mensaje *CONNECT*, que incluye información sobre sus IP públicas, a otro *peer B* a través del *Circuit Relay*, y pone en marcha un contador de tiempo para medir el RTT (Round-Trip Time). Cuando el *peer B* recibe este mensaje, envía a su vez un mensaje *CONNECT* al *peer A* que ha iniciado la conexión, que al recibir el mensaje *CONNECT* para el contador para medir el valor del RTT. Seguidamente, el *peer A* responde con un mensaje *SYNC,* y espera como máximo la mitad del valor del RTT previamente medido a que el *peer B* establezca una conexión directa con él usando una IP pública de las que contenía el primer mensaje *CONNECT* enviado por el *peer A*. Cuando se establece esta conexión, el *peer A* hace lo propio con el *peer B*, de modo que se puede afirmar que ambos nodos se sincronizan para establecer conexiones directas entre ellos en ambas direcciones. Debido a que realmente el mecanismo para establecer la conexión directa depende del protocolo de transporte utilizado (TCP o QUIC, por ejemplo), en el citado artículo [76] se

describen estas operaciones con más detalle. Por último, el *Circuit Relay* ya no es necesario después del establecimiento de una conexión directa entre *peers*, así que esta conexión puede cerrarse para liberar recursos en el nodo *relay* y que puedan ser aprovechados por otros *peers* de la red.

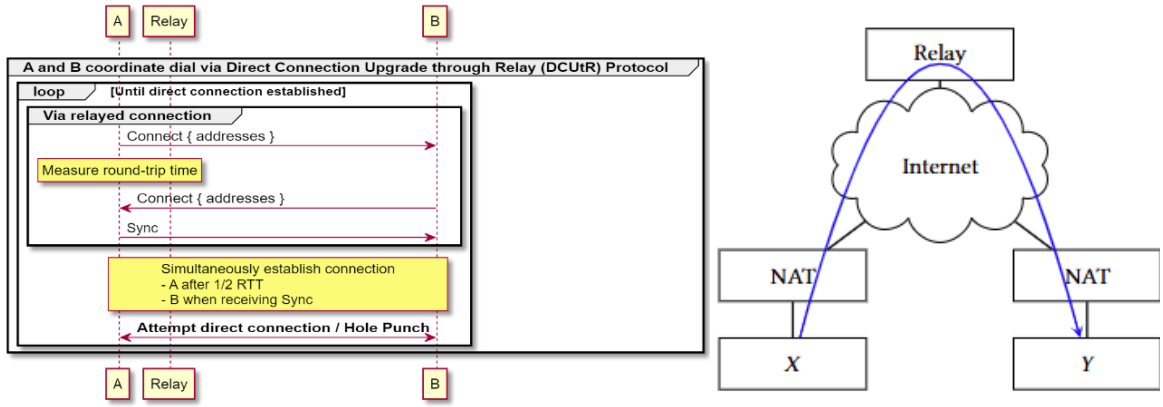


Figura 2.3: Proceso de conexión entre dos *peers* privados mediante el uso del protocolo DCUtR de *libp2p* [71][76]

Como se ha mencionado previamente, se realizó un primer experimento en el con voluntarios del equipo de Protocol Labs para medir el desempeño de *libp2p* en redes reales (residenciales y corporativas) para conectar directamente *peers* cuyos resultados se incluyeron en el artículo [76]. Se consiguió establecer una conexión directa en el 86% de los intentos para TCP y en el 93% para QUIC, unas tasas de éxito altas como se puede observar, pero además comprobaron que la gran mayoría de conexiones exitosas suceden en el primer intento y que los reintentos de conexiones fallidas no mejoran la tasa de éxito. El éxito de este experimento hizo que los desarrolladores publicaran abiertamente el código de estas pruebas [77] con el propósito de que usuarios independientes las llevaran a cabo en sus entornos y reportaran sus resultados para intentar mejorar esta tecnología.

En cuanto a la implementación real de *libp2p*, pueden citarse varios proyectos de cuyo núcleo tecnológico forma parte, aunque claramente destaca su integración en la tecnología Ethereum desde 2023 (especialmente su parte de mecanismos de publicación y suscripción distribuida) [78], que se trata de una plataforma abierta de *blockchain* que a su vez gestiona la criptomoneda *Ether*. También es interesante el proyecto EdgeVPN, una VPN totalmente descentralizada con descubrimiento de nodos automático que utiliza *libp2p* para proporcionar estas funcionalidades [79]. Además, EdgeVPN es utilizado a su vez por Kairos, un proyecto *sandbox* de la CNCF que se trata de "una meta-distribución Linux totalmente *cloud-native* para ejecutar Kubernetes", para poder crear una red P2P de nodos de K8s en entornos heterogéneos del edge computing con el objetivo final de formar un clúster de K8s [80]. Como muestra de la posible interacción entre las soluciones en la capa de red y de aplicación, Kairos ha explorado el uso de kube-vip (recordemos, basado en WireGuard) para conectar nodos del plano de control con el objetivo de crear clústeres con alta disponibilidad (*high availability* —*HA*), mientras que mantiene el

uso de *EdgeVPN* para añadir nodos *workers* a dichos clústeres [81]. Finalmente, es necesario mencionar el proyecto EdgeMesh de KubeEdge que se describirá con más detalle en el apartado 2.6.5.2 de esta tesis.

2.6. Virtualización utilizando contenedores

2.6.1. Evolución de la virtualización

Uno de los pilares fundamentales del éxito de la computación en la nube radica en su capacidad para virtualizar la infraestructura de computación disponible, al mismo tiempo que habilita la gestión eficiente de las cargas de trabajo (servicios o aplicaciones) desplegadas sobre este entorno virtualizado [82]. Las máquinas virtuales (VMs) fueron el primer intento de virtualización, permitiendo la creación de instancias reducidas de máquinas sobre la misma infraestructura física. Este paradigma dominó el ámbito de la computación en la nube durante varios años hasta que surgieron nuevos enfoques que buscaban resolver varias limitaciones de las VMs. En primer lugar, la sobrecarga resultante debido a la necesidad de instalar múltiples sistemas operativos para soportar diferentes VMs impedía que esos recursos se dedicaran a la ejecución de cargas de trabajo reales. De igual manera, la diversidad de los servicios desplegados fue aumentando paulatinamente (transmisión de video, procesamiento de datos de sensores, IA...), requiriendo un alto grado de dinamismo en la instalación y gestión de estos servicios, algo que los hipervisores de VMs no podían proporcionar [83]. Además, el uso de hipervisores requería que los propietarios y desarrolladores de sistemas tuvieran un conocimiento previo en cuanto los recursos necesarios y las características de la infraestructura que debían seleccionar para desplegar sus servicios o aplicaciones. Como consecuencia, emergieron varias iniciativas con el principal objetivo de superar estos problemas, cuya puesta en escena se tradujo en un gran éxito en el ecosistema del *cloud computing*.

Para intentar mitigar las sobrecargas causadas por la instalación de un OS huésped completo para cada máquina virtual, surgió una nueva técnica de virtualización conocida como virtualización en contenedores. Según IBM, los contenedores son "unidades ejecutables de *software* que empaquetan el código de una aplicación junto con las bibliotecas y dependencias necesarias para su ejecución" [84]. El principio rector de este tipo de virtualización radica en que no requiere de un hipervisor ni de un OS independiente por cada instancia; en su lugar, únicamente se virtualiza el OS de la máquina anfitriona, permitiendo que los contenedores se desplieguen de manera aislada, pero compartiendo el mismo *kernel* del *host* [85]. Este cambio de enfoque aporta una reducción significativa en el consumo de recursos, que se traduce en una mayor ligereza y en un aumento notable de su escalabilidad, en comparación con las VMs tradicionales.

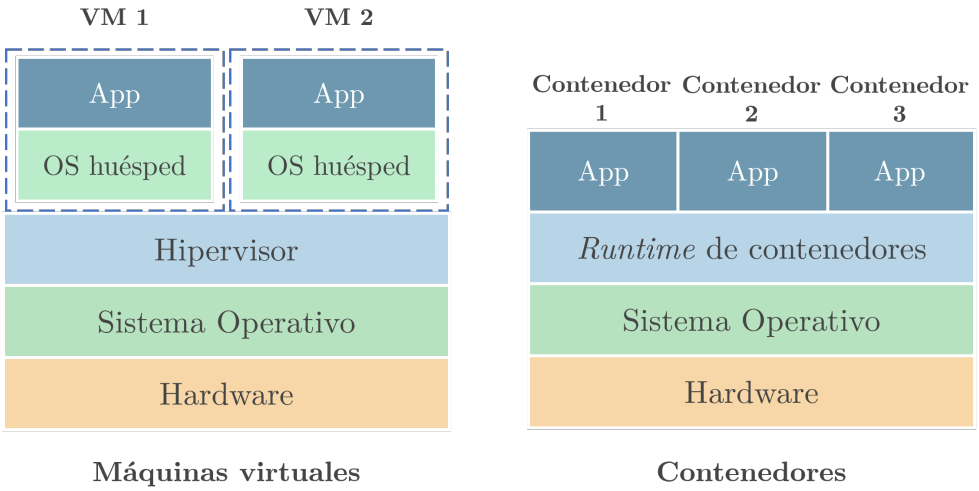


Figura 2.4: Comparación de las máquinas virtuales y los contenedores

Debido a todas estas ventajas, los contenedores se han convertido en el estándar *de facto* para ejecutar servicios virtualizados en la nube (*cloud-native*) y también son la evolución natural de los despliegues anteriores basados en VMs [86]. Aunque existen varias tecnologías de contenedores (como LXC [87] —LinuX Containers— que fue uno de los primeros *runtimes* de contenedores desarrollados), desde el lanzamiento de Docker [88] en 2013, esta tecnología fue ganando protagonismo hasta consolidarse como la solución estándar para esta técnica de virtualización. En consecuencia, Docker se ha convertido en el *container engine* (este concepto se detallará en el siguiente apartado) más utilizado, ya que ha sido probado con éxito y ampliamente adoptado en despliegues de grado de producción en las nubes públicas y privadas de la industria para diferentes propósitos.

Transición en la infraestructura de los operadores de telecomunicaciones

Esta evolución de las tecnologías de virtualización también ha tenido un gran impacto en la infraestructura de red de los operadores de telecomunicaciones, ya que ha permitido virtualizar las funciones de red (NFV) facilitando su despliegue en equipamiento de propósito general (servidores o máquinas al uso). Este avance permite desacoplar los servicios provistos (parte *software*: enrutamiento, funciones de cortafuegos...) del *hardware* específico que los ejecuta o provee (*routers*, cortafuegos, *switches*...), y, por consiguiente, reducir los costes de operación y mantenimiento de toda esta infraestructura.

En este ámbito, las funciones de red virtualizadas (VNFs, por ejemplo, las plataformas de código abierto libres de cortafuegos y enrutamiento pfSense [89] y OPNsense [90]) empezaron a desplegarse como VMs, por lo que estas funciones de red terminaban siendo desarrolladas siguiendo un patrón de aplicación monolítica. Este patrón de diseño presenta claras desventajas, como la necesidad de modificar la VNF por completo para aplicar cualquier cambio mínimo (solución de errores,

implementación de nuevas funcionalidades o parches de seguridad, etc.), o la sobrecarga que añaden las propias VMs, que es aún más limitante en entornos donde se despliegan una gran cantidad de VNFs diferentes, resultando en altos consumos de memoria y el cuestionamiento del beneficio en calidad de servicio al cliente final de la operadora. En este caso, la necesidad de evolucionar este tipo de infraestructura hacia un entorno *cloud-native* se hizo todavía más patente, por lo que las VNF se empezaron a virtualizar utilizando contenedores, pasando a conocerse como CNF (Cloud-Native Network Functions). Este proceso se ilustra en la Figura 2.5.

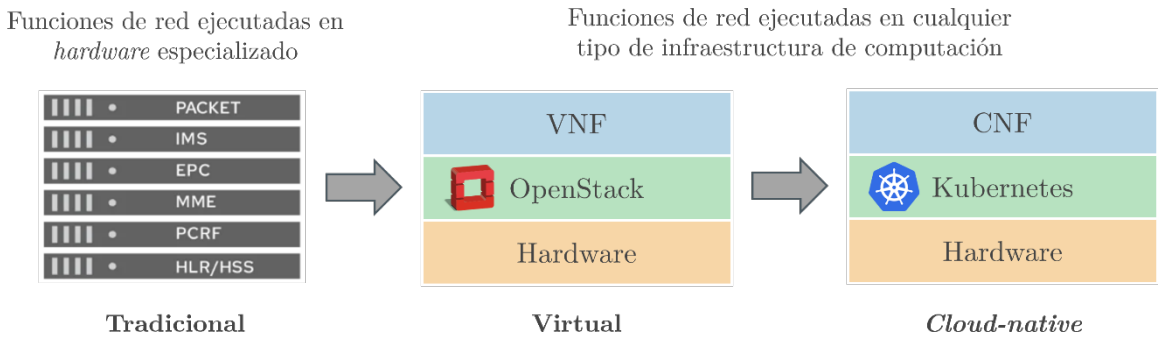


Figura 2.5: Evolución de la virtualización de las funciones de red

Asimismo, abrazar completamente el paradigma de diseño *cloud-native* va más allá del uso de contenedores, ya que conlleva asociado múltiples beneficios y funcionalidades como microservicios, *service mesh*, observabilidad y el uso de tecnologías avanzadas de orquestación como Kubernetes (K8s, se describirá en detalle en el próximo punto 2.6.5.1) [91]. Siguiendo esta línea, la evolución de VNFs a CNFs no es tan trivial como el simple reemplazo de máquinas virtuales y controladores de los recursos de virtualización del *hardware* (como la herramienta de código abierto OpenStack [92]) por contenedores y orquestadores de contenedores, como Kubernetes. El principal obstáculo para la transición hacia las CNFs es la gran cantidad de VNFs existentes, algunas con funcionalidades no extrapolables o que conllevaría un gran esfuerzo de migración. Además, los contenedores son, por definición, más inseguros que las VMs (aunque en los últimos años se han realizado numerosos esfuerzos para mitigar esta problemática) y presentan un menor aislamiento respecto a la máquina anfitriona en la que se ejecutan en comparación con las VMs [91].

Para intentar mitigar temporalmente esta problemática en la migración de VNFs desde VMs a contenedores, se han desarrollado algunas herramientas (entre las que destaca KubeVirt [93]) que permiten la ejecución de VMs directamente manejadas por K8s, quedando integradas en este sistema de orquestación como si se trataran de un contenedor más. De esta manera se permite avanzar hacia el uso de ecosistemas *cloud-native,* aunque se sigan utilizando VNFs para casos específicos, conviviendo sin interferencias con las nuevas CNFs.

Finalmente, se han diseñado estándares con el propósito de facilitar la creación de herramientas para supervisar el ciclo de vida de estas funciones de red virtualizadas, tomando el liderazgo el ISG NFV de la ETSI a través de los estándares NFV Management and Orchestration (MANO) [94]. Esta especificación ha tenido un alto grado de aceptación tanto en el ámbito investigador como en el comercial debido a las grandes posibilidades que proporciona para el desarrollo de nuevas aplicaciones y modelos de negocio. A nivel funcional, destaca la implementación Open Source MANO (OSM) también desarrollada por la ETSI, y que es totalmente compatible con el estándar anterior [95].

2.6.2. Contenedores: estandarización y funcionamiento

La Open Container Initiative (OCI) fue fundada en 2015 por las compañías líderes y sus expertos en la tecnología de virtualización utilizando contenedores (Docker, CoreOS...) con el objetivo de especificar un estándar para esta tecnología, debido al potencial que encontraron en ella. Finalmente, esta acción resultó en la creación de tres especificaciones: la Especificación de *Runtime* (*runtime-spec*, la especificación para ejecutar contenedores), la Especificación de Imagen (*image-spec*, para empaquetar contenedores en imágenes) y la Especificación de Distribución (*distribution-spec*, para compartir y utilizar imágenes de contenedores) [96]. Dos años después, cuando el Docker Engine [97] se consolidó como la referencia tecnológica para ejecutar contenedores, Docker hizo público el código de su plataforma de contenedores, dividiéndolo en módulos (incluyendo su *container runtime* de alto nivel —containerd [98]— y su *container runtime* de bajo nivel —*runC* [99]), con el propósito de ofrecer a los desarrolladores de sistemas la capacidad de construir sistemas basados en contenedores totalmente personalizados. Esta innovación recibió el nombre de Proyecto Moby [100], la cual permitió que los ingenieros pudieran centrarse en el desarrollo de *runtimes* de contenedores más ligeros mediante la reducción de sus componentes internos y la supresión de módulos innecesarios. De esta manera se puede aumentar su rendimiento o incluso diseñar sistemas operativos específicos que incorporen estos *runtimes*, tendencias que resultan interesantes tanto para el *cloud* como para el *edge*. En consecuencia, las subsecciones siguientes profundizan en estos dos enfoques y en las iniciativas identificadas para escenarios de computación en el *edge*.

Los *container engines* (por ejemplo, Docker o Podman [101]) están compuestos por diferentes herramientas que tienen el objetivo final de construir y ejecutar imágenes de contenedores. Para lograr esto último, un *runtime* de contenedores de alto nivel (*high-level runtime,* también conocido como gestor de contenedores, este bloque controla el transporte y la gestión de las imágenes de contenedores, que se descargan de los registros o repositorios de imágenes) interactúa con su homólogo de bajo nivel (el bloque que recibe la imagen de contenedor desempaquetada del *runtime* de alto nivel y finalmente ejecuta el

contenedor en la máquina, interactuando directamente con el *kernel* del sistema operativo anfitrión), como se muestra en la Figura 2.6. Como se describe en la introducción a esta sección, el *runtime* del Docker Engine (estándar *de facto*) se materializa en la interacción de los *runtimes* containerd y runC. Sin embargo, existen otras implementaciones de *runtimes* que se han desarrollado teniendo en cuenta las limitaciones presentes en los equipos vinculados con el paradigma del *edge computing*, las cuales se detallarán en la siguiente sección.

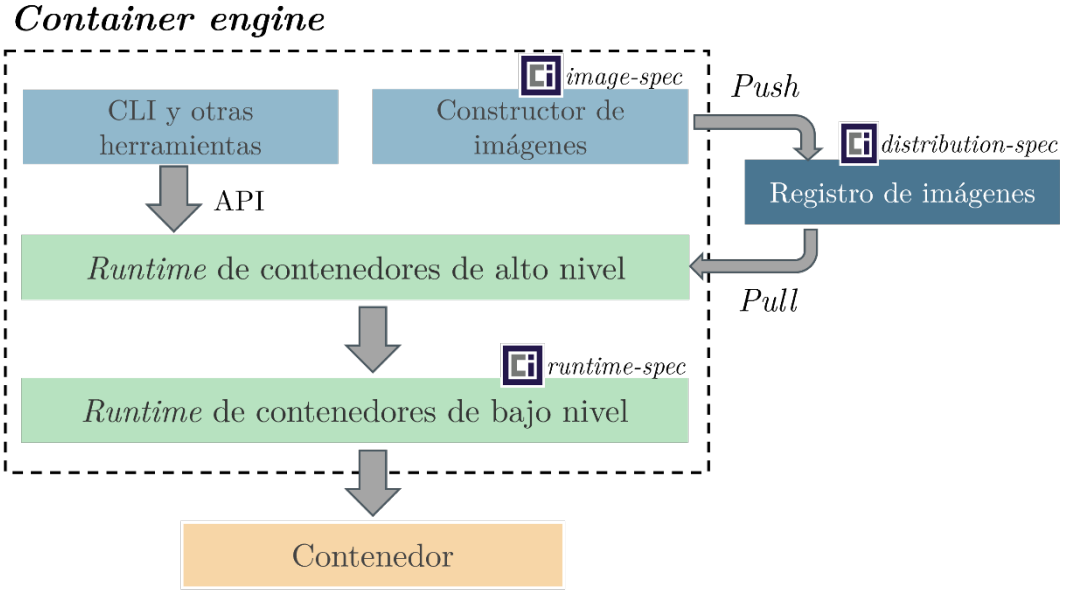


Figura 2.6: Funcionamiento de un *container engine*

A modo de ejemplo funcional, en el caso de Docker, un usuario define la imagen de contenedor en un fichero llamado *Dockerfile* utilizando un lenguaje de etiquetado específico. Esta imagen se compila (a partir del código fuente de un desarrollo *software*) cuando se ejecuta el comando *docker build* del Docker CLI (que cuenta con soporte para la creación de imágenes multi arquitectura, es decir, una misma imagen que puede ser ejecutada en máquinas con diversas arquitecturas de CPU: x86_64, ARM64, RISC-V...), y puede almacenarse en un registro de contenedores gracias al comando *docker image push*. Para ejecutar un contenedor, se utiliza el comando *docker run*, el cual realmente realiza una llamada a containerd (*runtime* de alto nivel), para que este, a su vez, transmita la orden de ejecución al *runtime* de bajo nivel runC.

Además, Docker proporciona una herramienta adicional llamada Docker Compose para facilitar el despliegue simultáneo de varios contenedores que cuenten con una relación de dependencia entre ellos (por ejemplo, una API junto con su base de datos). Este despliegue se puede configurar (dependencias entre contenedores, apertura de puertos, redes internas, uso de volúmenes persistentes...) en un fichero en formato YAML. Por lo tanto, se podría afirmar que este tipo de despliegues fue el primer paso hacia la orquestación de contenedores, un concepto que se describirá con más detalle en la sección 2.6.5.

2.6.3. *Runtimes* para la ejecución de contenedores en el *edge*

Como se ha indicado previamente, las limitaciones de *hardware* de los dispositivos empleados en el *edge computing* pueden representar una barrera significativa para la ejecución del Docker Engine. Por lo tanto, dentro del marco de la adaptación de los estándares *cloud-native* a *edge-native*, se han desarrollado diferentes *runtimes* alternativos teniendo en cuenta estas restricciones presentes en los dispositivos del ámbito del *edge computing*. El objetivo de esta subsección es realizar un estudio de las alternativas a Docker más relevantes para estos entornos.

crun es un *runtime* de contenedores de bajo nivel totalmente compatible con la especificación de *runtime* de la OCI, desarrollado en C, al igual que LXC y el *kernel* de Linux, y a diferencia de runC, que está escrito en Go. En este sentido, crun mejora a runC, ya que C puede ofrecer un mejor rendimiento que Go para este propósito. Además, el binario de crun presenta un tamaño considerablemente más reducido que el de runC (300 KB frente a 15 MB). De hecho, en algunos estudios crun ha demostrado un mejor rendimiento, como en el caso de una prueba que consistió en ejecutar secuencialmente 100 contenedores. En esta prueba, crun alcanzó el objetivo en 1,69 s, mientras que runC necesitó 3,34 s [102].

Rust es un lenguaje de programación que lanzó su versión 1.0 en 2020; desde entonces, se ha establecido como una fuerte alternativa a C y C++ debido a sus notables mejoras en la seguridad en la gestión de la memoria. Debido a que Rust también supera a Go en la implementación de las llamadas al sistema, se utilizó para desarrollar *youki*, un *runtime* de contenedores de bajo nivel compatible con el estándar OCI que, al igual que crun, se caracteriza por un binario de tamaño reducido, permitiendo que ambos sean adecuados para entornos con recursos limitados. En una prueba similar a la realizada para crun, youki demostró un mayor rendimiento en comparación con runC, mientras que crun siguió siendo el más rápido [103].

En cuanto a los *runtimes* de alto nivel, CRI-O emerge como una de las soluciones más interesantes. Se trata de un *runtime* de contenedores ligero diseñado específicamente para Kubernetes tal y como reza su descripción principal ("una implementación del Container Runtime Interface —CRI— de Kubernetes [104] basada en el estándar OCI"), por lo que no está diseñado para ser ejecutado fuera del ecosistema de K8s, ya que su propósito es ejercer de elemento integrador entre *runtimes* de bajo nivel y el componente *kubelet* de K8s [105]. Su principal ventaja es una reducción del consumo de recursos en comparación con el containerd de Docker, Moby o *rkt*, y su compatibilidad con cualquier *runtime* de bajo nivel que siga el estándar OCI, una característica interesante que permite a este *runtime* ejecutar cargas de trabajo no solo basadas en contenedores, una tendencia prometedora que se describe con más detalle en la sección 2.7 de este capítulo sobre

el estado del arte. De hecho, a partir de su versión 1.31 (lanzada en septiembre de 2024) utiliza el previamente descrito crun como *runtime* de bajo nivel por defecto [106]. Sin embargo, CRI-O no ha sido desarrollado para su uso en dispositivos embebidos de bajos recursos por el requerimiento de diseño anteriormente comentado de que el dispositivo anfitrión sea capaz de ejecutar Kubernetes.

Otra iniciativa interesante es iSulad [107], que pretende competir tanto con Docker como con containerd, a nivel de *engine* y de *runtime* de alto nivel, respectivamente. Esta iniciativa forma parte del OS de código abierto openEuler [108], que a su vez está gestionada por la OpenAtom Foundation, la primera fundación para la promoción de iniciativas de código abierta de China. A nivel técnico, iSulad está desarrollado en C y C++ con el gran objetivo de ser un *runtime* de alto nivel ligero y soportado por múltiples arquitecturas que pueda ejecutarse en equipos relacionados con el IoT. Al igual que CRI-O, soporta la especificación CRI (Container Runtime Interface), por lo que también es compatible con K8s, y diferentes *runtimes* de bajo nivel.

Finalmente, una ventaja competitiva con la que cuenta containerd es la herramienta nerdctl (contaiNERD CTL), un CLI que pretende ser el equivalente de Docker Compose para containerd, incluso soportando especificaciones con el mismo formato, por lo que permite ejecutar directamente archivos del tipo *docker-compose.yaml* [109]. De esta manera, se habilita la ejecución de contenedores sobre containerd sin necesidad de instalar en entornos *edge* toda la sobrecarga adicional que conlleva una instalación completa del Docker Engine.

2.6.4. Sistemas operativos específicos para la ejecución de contenedores

De la misma manera que el tamaño y los requerimientos de ejecución de los *container runtimes* se están reduciendo para adaptarse a un rango más amplio de sistemas, en los últimos años han aparecido algunos sistemas operativos específicos que aprovechan la mayor ligereza y portabilidad de estos *runtimes* con el objetivo principal de llevar la virtualización de contenedores a dispositivos que cuentan con menos recursos de computación, como algunos sistemas embebidos o SBCs.

EVE-OS es un sistema operativo desarrollado originalmente por ZEDEDA y luego donado bajo una licencia de código abierto a la Fundación Linux, que lo ha incluido en su pila de proyectos de investigación sobre el *edge computing* (LF-Edge) [18]. El objetivo principal del desarrollo de este sistema operativo es proporcionar a los dispositivos de computación en el *edge* un sistema operativo con una arquitectura universal, independiente del proveedor y estandarizada, siguiendo la misma estrategia que utilizó Google en el mercado de móviles inteligentes al lanzar Android. Además, EVE-OS ha adoptado proyectos de código abierto reconocidos

como Xen Project, LinuxKit y Alpine Linux para su desarrollo. Actualmente, es posible gestionar remotamente una flota de dispositivos que ejecutan EVE-OS bajo un modelo de seguridad *zero trust* utilizando el controlador Adam, que es la implementación de referencia de un controlador compatible con la API de LF-Edge. Igualmente, EVE-OS proporciona soporte para ejecutar cargas de trabajo virtualizadas en contenedores utilizando containerd y para ejecutar distribuciones de Kubernetes sobre él. En consecuencia, los contribuidores de EVE anunciaron una propuesta de arquitectura para integrar EVE con Kubernetes a nivel del plano de control que aún no se ha materializado [110]. Este sistema operativo está diseñado para dispositivos del *edge* con capacidades de computación razonables (un mínimo de 512 MB de memoria RAM) y ha sido preparado para ser desplegado en una amplia gama de arquitecturas de CPU. Asimismo, EVE-OS tiene las suficientes funcionalidades para ser desplegado en *bare metal* (directamente sobre el *hardware* de la máquina, sin sistemas operativos o *software* de virtualización adicional) y soporta una gran variedad de tecnologías para el despliegue de aplicaciones que se pueden combinar entre ellas: VMs, contenedores Docker y clústeres de Kubernetes.

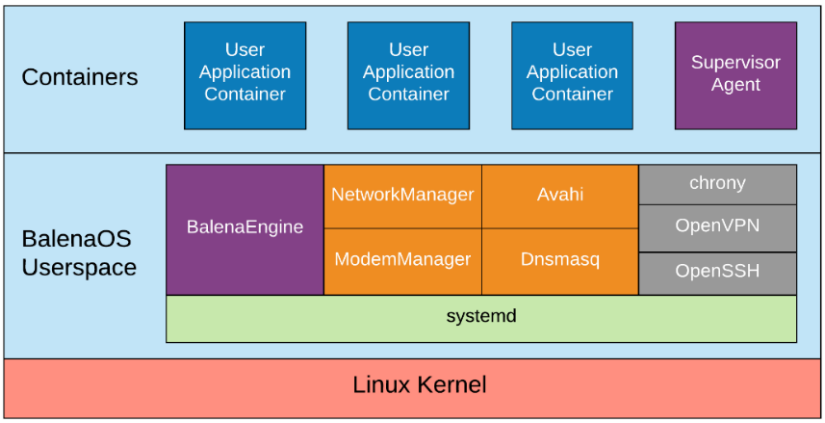


Figura 2.7: Arquitectura de bloques de balenaOS [111]

balenaOS es un sistema operativo anfitrión cuyo propósito es ejecutar contenedores Docker en dispositivos embebidos, habiendo sido diseñado para soportar condiciones de red y de funcionamiento inestables y discontinuas [111]. Este sistema operativo se basa en el Yocto Project, un proyecto que facilita la creación de OS basados en Linux para sistemas embebidos [112], por lo que tiene un impacto limitado en el uso de memoria y ofrece la posibilidad de ser fácilmente portado a dispositivos más potentes, estando disponible para una gran variedad de arquitecturas de CPU [113]. Su propósito es replicar la estrategia de despliegue de aplicaciones de los sistemas operativos en la nube en dispositivos propios del *edge computing* mediante el uso de contenedores (transición de *cloud-native* a *edge-native*). Para ello, Balena ha desarrollado su propio *container engine* (balenaEngine [114]) basado en la tecnología del proyecto Moby. Este *engine* fue adaptado mediante: (i) la eliminación de las funciones más demandantes de Docker orientadas a la nube (Swarm, soporte de *plugins*, controladores de redes *overlay*...) que no son necesarias en los nodos en el *edge*; y (ii) la adición de características específicas para dispositivos de computación menos potentes (soporte para las arquitecturas de CPU

más comunes en computadoras monoplaca – conocidas como *single-board computers* o SBCs, mejora en la descarga de imágenes de contenedores para prevenir fallos de red, etc.). Actualmente, el sistema operativo es compatible con hasta 50 tipos de dispositivos [115].

La empresa Balena también ofrece Balena Cloud como servicio, una plataforma automatizada para gestionar toda la infraestructura de los usuarios que ejecutan balenaOS y las cargas de trabajo desplegadas en dichos dispositivos, optimizada para el *edge*. Además, a pesar de tratarse de un producto comercial, Balena ha publicado este *software* de gestión bajo una licencia de código abierto (llamado como OpenBalena) para que pueda ser mejorado por la comunidad, o simplemente para que los usuarios puedan alojar esta plataforma en su propia infraestructura sin depender de Balena y, por lo tanto, no pagar por su uso [116]. Mediante la utilización de esta plataforma, los desarrolladores pueden desplegar aplicaciones empaquetadas en contenedores, enviar actualizaciones y monitorizar el estado de la flota formada por los dispositivos previamente registrados.

Según la empresa Pantacor, Docker fue diseñado sin considerar los dispositivos embebidos, ya que requiere una alta disponibilidad de recursos. En consecuencia, desarrollaron Pantavisor para relajar esos requisitos. Éste se trata de un *runtime* de contenedores de bajo nivel escrito en C que comparte similitudes con crun (estando más cercano al *kernel* de Linux) [117]. Pantavisor tiene como propósito transformar los sistemas tradicionales en un conjunto de microservicios portátiles (materializados en contenedores Linux) en lugar del esquema tradicional que consiste en el uso de *firmware* y aplicaciones. Según Pantacor, Pantavisor "está pensado para ser un sistema de arranque compuesto por un solo binario que arranca directamente desde el *kernel* y se convierte en el primer proceso en ejecutarse, el cual luego pone en marcha el resto del sistema como un conjunto de micro contenedores bien definidos" [118]. Este *runtime* de contenedores es compatible con las arquitecturas de CPU ARM, MIPS, RISC-V, x86 y PowerPC, consume una cantidad de memoria de aproximadamente 350 KB y requiere un mínimo de solo 64 MB de RAM.

A diferencia de las arquitecturas tradicionales basadas en contenedores, Pantavisor no necesita un *runtime* de contenedores que se ejecute sobre un sistema operativo completo, resultando en un patrón más adecuado para dispositivos embebidos, ya que estos podrían no utilizar todas las funciones de un OS anfitrión. De esta manera, Pantavisor virtualiza en contenedores la capa del sistema operativo anfitrión, incorporando características propias de los contenedores como la actualizabilidad y escalabilidad, entre otras. Estos contenedores son ejecutados por el *runtime* de Pantavisor, que es único presente en el sistema.

Al igual que Balena, Pantacor proporciona un marco (PantacorHub) para gestionar los dispositivos que ejecutan Pantavisor y los servicios desplegados en ellos. Es necesario destacar que éste se ofrece tanto como herramienta de código abierto para ser gestionado de manera privada o como un servicio de pago alojado por la propia empresa.

2.6.5. Orquestación de contenedores

2.6.5.1. Kubernetes

La gestión de los contenedores mediante el uso directo de *runtimes* y *engines* es perfectamente posible, puesto que, como se ha visto en la anterior sección 2.6.2, ofrecen una serie de herramientas para dicho propósito, como APIs o CLIs. Sin embargo, estas operan de manera individual para cada contenedor o sobre agrupaciones reducidas, por lo que presentan varias limitaciones que se hacen más evidentes en cuanto aumenta el número de contenedores a gestionar. Este sería el caso de una entidad que tuviera que realizar tareas como el pausado de la ejecución o la actualización de las versiones de sus servicios, ya que resultaría inviable realizar dichas acciones de manera individual para cada uno de ellos.

Como intento de poner solución a estas limitaciones, han ido apareciendo algunas herramientas, que se ejecutan en una capa superior a las tecnologías de virtualización de contenedores, para gestionar y orquestar mejor su despliegue, permitiendo la programación, escalado y control de los contenedores. Kubernetes (K8s), actualmente gestionado directamente por la CNCF, ha ido superando a sus competidores con el paso de los años, como Docker Swarm o Apache Mesos [6][119], hasta convertirse en el estándar para la orquestación de contenedores y microservicios en la nube. En consecuencia, unido al hecho que K8s es una tecnología altamente personalizable y de código abierto (desarrollada en Go), la mayoría de los proveedores de nube han implementado sus propias distribuciones de K8s con el objetivo de optimizar la integración de K8s con sus sistemas y tecnologías propietarias. Por ejemplo, AWS ofrece clústeres de K8s utilizando EKS (Elastic Kubernetes Service) [120], de igual manera que Red Hat utiliza OpenShift Kubernetes Engine [121]. Estas llamadas distribuciones de K8s han de cumplir completamente con el estándar de K8s para poder estar certificadas por la CNCF [122], lo cual significa que son agnósticas para el usuario desde el punto de vista de la interacción con un clúster. De hecho, no existe ningún método estandarizado para obtener la distribución de K8s que está utilizando un clúster.

Respecto a la ejecución de contenedores en Kubernetes, la unidad mínima para su despliegue son los llamados *Pods.* Un *Pod* puede contener uno o varios contenedores, los cuales comparten sistema de ficheros y *namespace* de red. Es sobre esta unidad mínima donde K8s añade unos elementos de control más complejos, que se diferencian en la manera de desplegar dichos *Pods.* Los tipos básicos son: *Deployment* (es el tipo más básico que se utiliza normalmente en aplicaciones sin estado), *StatefulSet* (despliega *Pods* de manera ordenada, especialmente útil para aplicaciones con estado), *DaemonSet* (despliega una réplica del *Pod* en cada nodo del clúster), *Job* (despliega un *Pod* pensado como una tarea única que se tiene que ejecutar un número determinado de veces), *CronJob* (ejecuta *Jobs* con una periodicidad temporal). Además, K8s proporciona unos elementos

llamados *Services* para configurar la manera de exponer una aplicación (definición de los puertos de red...), de manera que actúa como un *load balancer* sobre los *Pods* desplegados.

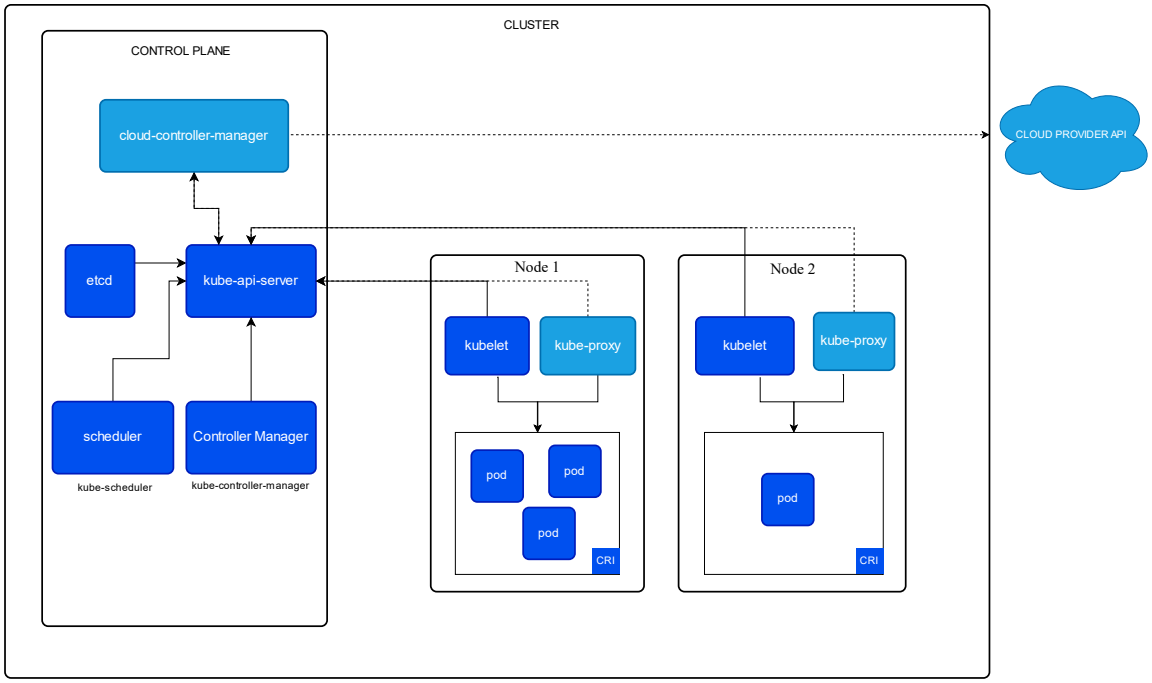


Figura 2.8: Arquitectura de un clúster de Kubernetes [123]

La arquitectura de Kubernetes a alto nivel se divide en diversos clústeres, que a su vez están formados por varios nodos, los cuales podrían describirse como el equivalente a una máquina física, aunque suelen tratarse de máquinas virtuales en los entornos de la nube [123]. Estos nodos pueden clasificarse en:

- **Nodos del plano de control:** anteriormente llamados nodos *master*, ejecutan el llamado plano de control del clúster, que está formado por los elementos *kube-apiserver* (expone la API de K8s), *kube-scheduler* (decide el nodo en el que se despliegan los contenedores en forma de *Pods*), *kube-controller-manager* (ejecutan los llamados *controller processes* para controlar los diferentes tipos de elementos de un clúster, como los nodos, *Jobs*, etc.) y *cloud-controller-manager* (este elemento es opcional, permite enlazar el clúster con una API específica de un proveedor en la nube, especialmente útil para entornos *cloud-native*). Además, contienen la base de datos *etcd* —tecnología NoSQL— de tipo clave valor (en inglés *key-value databases)* y es una de las partes fundamentales de un clúster, debido a que persisten su información esencial, como la información de los nodos (arquitectura de CPU, direcciones IP...) o de los servicios desplegados en cada uno de ellos. Debido a que estos nodos contienen tienen un papel esencial en el funcionamiento del clúster, pueden ser replicados para alcanzar un escenario de alta fiabilidad o *high availability* (HA).

- **Nodos *workers*:** se limitan a la ejecución de contenedores (encapsulados en *Pods*), por lo que son controlados por los nodos del plano de control. Es necesario que un clúster cuente con al menos un nodo de este tipo para poder ejecutar contenedores. Además, este tipo de nodos ofrece una flexibilidad que por definición no pueden tener los nodos del plano de control. Esto significa que puede reemplazarse con facilidad, así como escalarse dinámicamente. A nivel interno, cuentan con los elementos *kubelet* (su misión es asegurarse de que los contenedores se ejecutan dentro de *Pods*) y *kube-proxy* (*proxy* de red para controlar la exposición de los servicios desplegados, es un elemento opcional que puede ser reemplazado por un plugin de red).

A nivel de usuario, K8s proporciona por defecto el CLI *kubectl* [124] que permite interactuar con el plano de control del clúster mediante la API del *kube-apiserver* y realizar las acciones oportunas en el mismo. Sin embargo, esta herramienta se trata realmente de la manera más básica de interactuar con un clúster, dado que requiere un cierto nivel de conocimiento previo de los recursos y el funcionamiento operacional de K8s. Es más, kubectl no proporciona mecanismos avanzados de abstracción o automatización, ni tampoco una interfaz gráfica que facilite la gestión para usuarios menos experimentados.

Finalmente, es necesario aclarar que, aunque se haya descrito el funcionamiento básico de K8s de manera breve pero concisa, se trata de una tecnología muy compleja con una curva de aprendizaje pronunciada, por lo que indagar en detalle en su funcionamiento conllevaría un esfuerzo para crear un contenido que, a juicio del doctorando, va más allá del objetivo de esta tesis doctoral, y que está disponible en numerosas fuentes de información, como la propia documentación oficial de K8s.

2.6.5.2. Alternativas a Kubernetes adaptadas al *edge*

En los subapartados anteriores ha quedado patente que es posible utilizar contenedores en los despliegues propios del *edge computing*. Por lo tanto, el siguiente paso lógico consiste en encontrar una tecnología estandarizada de orquestación de contenedores adaptada al *edge computing*, intentando transferir los atributos *cloud-native* proporcionados por la herramienta que define el estándar *de facto* para la orquestación de contenedores: Kubernetes.

Esta no es una acción trivial, ya que Kubernetes por sí mismo se encuentra lejos de ser solución adecuada. Aunque su despliegue es factible en algunos nodos de computación a lo largo del continuo [125][126] (aquellos que pueden soportar cargas de trabajo más demandantes), esta tecnología no se adapta a las capacidades de equipos más limitados en recursos como pueden ser nodos del *far-edge* o nodos finales (también conocidos como *leaf nodes*). Algunos trabajos (por ejemplo,

[127][128]), han estudiado profundamente este problema al mismo tiempo que sugieren algunas alternativas. La intención de esta sección es precisamente analizar posibles candidatos para convertirse en la alternativa a K8s para la orquestación de contenedores en el ámbito del *edge computing*.

Se puede afirmar que las soluciones que se han encontrado en trabajos de investigación relevantes, así como en repositorios de código y tecnologías con licencias abiertas, se pueden dividir en dos grupos principales. El primer grupo pone el foco en la adaptación de la arquitectura de Kubernetes para entornos propios del *edge* mediante la reducción del impacto en el uso de memoria de esta plataforma de orquestación, con el propósito final de ampliar la gama de equipos en los que K8s pueda ser desplegado funcionalmente. En esta línea, las diferentes distribuciones ligeras de Kubernetes pueden coexistir con un despliegue completo de K8s en la nube, creando una especie de entorno *multi-cluster* para lograr la comunicación y orquestación de contenedores en el continuo de computación. Este enfoque se ilustra en el esquema izquierdo de la Figura 2.9. Por otro lado, el esquema derecho de la misma figura representa la segunda tendencia, que consiste en la creación de nuevos *frameworks* específicamente adaptados al *edge*, de manera que se sigan los principios de Kubernetes, pero que a su vez no sean simplificaciones directas o reducciones de esa tecnología. Estas soluciones adaptan K8s a las características específicas del *edge computing* (conectividad de red inestable, dificultad para gestionar equipos heterogéneos y con pocos recursos...), manteniendo todos los beneficios de esta tecnología y sin limitarse simplemente a crear otra distribución ligera de K8s que no alcance una verdadera sinergia *edge-cloud*.

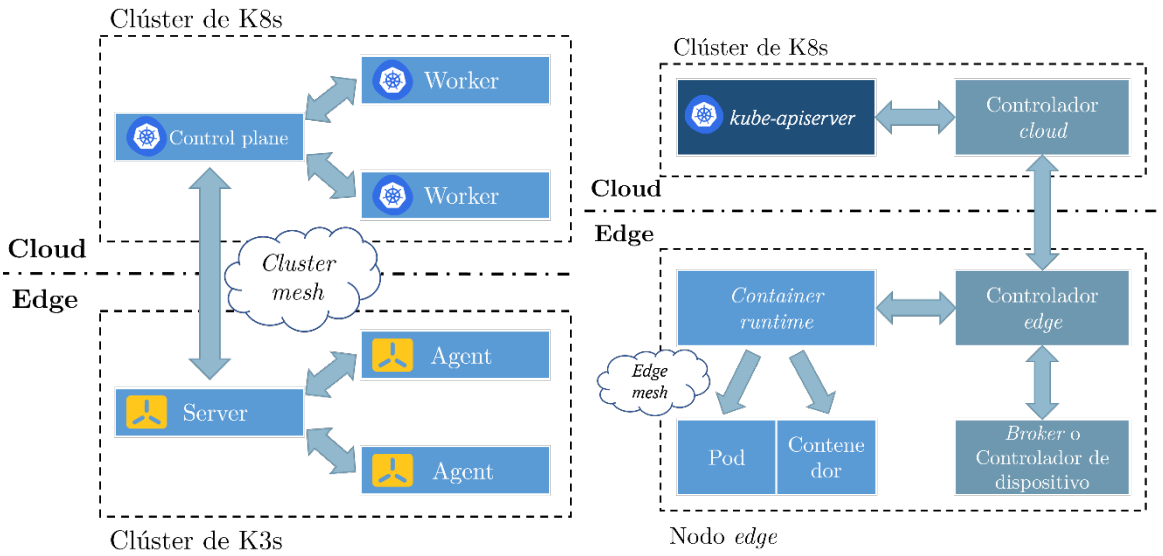


Figura 2.9: Comparación de arquitecturas para el despliegue de K8s en el *edge*: (i) Arquitectura de K8s *multi-cluster* utilizando distribuciones de K8s ligeras; (ii) K8s adaptado a los requerimientos del *edge computing*

Distribuciones ligeras de K8s

Dentro del primer grupo de soluciones destaca K3s, que se trata de una distribución ligera de K8s certificada por la CNCF, desarrollada por la empresa Rancher. Fue creada para ejecutarse en dispositivos menos potentes, con un impacto en memoria mucho menor que otras distribuciones de K8s disponibles en el mercado [129]. K3s modifica la arquitectura de alto nivel de un clúster de K8s transformando los nodos del plano de control y *workers* en nodos *server* y *agent*, con atribuciones similares, pero con algunas diferencias estratégicas. Además, ofrece tres posibles arquitecturas: (i) un solo servidor con una base de datos SQLite embebida; y (ii) servidores de alta disponibilidad utilizando una base de datos *etcd* embebida o (iii) una base de datos externa (basada en SQL o etcd). Esta distribución también está optimizada para varias arquitecturas de CPU, como ARM32, ARM64 y ARMv7. Por lo tanto, se trata de una opción muy recomendable para entornos propios del *edge* (que a menudo usan dispositivos embebidos, como Raspberry Pis [130] o placas NVIDIA Jetson [131], para que actúen como nodos del clúster). Los requisitos mínimos de K3s son 256 MB de RAM para un nodo *agent* y 512 MB para un nodo *server* con algunos servicios desplegados en el nodo *agent*. Asimismo, Rancher también desarrolló un sistema operativo para nodos del *edge* optimizado para ejecutar K3s, manteniendo solo los recursos mínimos del OS subyacente: k3OS [132], que posteriormente fue integrado en el OS *cloud-native* Rancher Elemental [133]. Su reducido impacto en memoria junto con su capacidad de ejecutarse en dispositivos con arquitecturas de CPU diversas convierte a K3s en la distribución de K8s más recomendable para construir clústeres en el *edge* siguiendo el primer enfoque.

Como una distribución ligera alternativa de K8s, Canonical lanzó MicroK8s, que afirma tener un uso mínimo de memoria de alrededor de 540 MB, pero su memoria recomendada es de 4 GB, que es considerablemente mayor que la de K3s [134]. MicroK8s se trata claramente de una tecnología menos madura y afianzada que K3s, que, según la experiencia personal del doctorando, aún no está lista para considerarse una opción robusta para despliegues puramente nativos del *edge* [135]. Sin embargo, MicroK8s tiene la ventaja de ser una distribución fácilmente personalizable, ya que ofrece una serie de complementos cuya instalación consiste simplemente en la ejecución de unos simples comandos. Los complementos disponibles incluyen algunos de los módulos de K8s más utilizados, como CoreDNS, Helm o Istio, y la posibilidad de lograr funcionalidades de HA de K8s de manera sencilla. Estas características hacen que MicroK8s sea una de las distribuciones de K8s más interesantes para desarrollo y *testing* en dispositivos *edge* ligeramente más potentes (especialmente aquellos con más de 1 GB de RAM). Estas dos distribuciones enfocadas al *edge* han sido comparadas entre sí y con una distribución completa de K8s en algunos trabajos como [136] y [137], mostrando una clara mejora de rendimiento en comparación con una distribución clásica, y sin encontrar diferencias de rendimiento significativas entre K3s y MicroK8s. Por lo tanto, se puede afirmar que aún no existe una distribución de K8s para el *edge computing*

estándar *de facto*, pero K3s cuenta con mayor popularidad, mayor robustez y un menor impacto en el uso de memoria.

Otra opción interesante es K0s, una distribución ligera de K8s también certificada por la CNCF que comparte muchas decisiones de diseño y operatividad con K3s, dado que ofrece diferentes arquitecturas de funcionamiento que varían dependiendo del tipo de base de datos utilizado para persistir la configuración del clúster (etcd embebida o externa; o basada en SQL externa) y de la necesidad de mecanismos de resiliencia [138]. Además, como su consumo se memoria es similar, su principal diferencia es que K3s está más establecido en la comunidad, contando con mayor participación y popularidad en su repositorio público de GitHub.

Finalmente, *kind* una opción a considerar para la puesta en marcha de un clúster con la finalidad de ser utilizado para las etapas de desarrollo. Esta herramienta ofrece la puesta en marcha de un clúster de K8s en una única máquina mediante el despliegue de contenedores de Docker para que actúen como los nodos del clúster. No obstante, esta es a su vez su principal limitación para ser considerada como una alternativa válida para el *edge*, ya que no soporta el uso de *container engines* alternativos como los que se han descrito en el apartado 2.6.3.

Soluciones basadas en K8s adaptadas al edge

En cuanto al segundo grupo, el diseño y la arquitectura de las diferentes tecnologías difiere ostensiblemente. El diseño funcional de gran parte de estos recientes *frameworks* que adaptan los principios de K8s al *edge* consiste en mantener el plano de control del sistema en la parte *cloud* (es decir, la parte centralizada y con mayores recursos de computación), y trasladar al *edge* (a la parte descentralizada o remota) solamente los servicios desplegados por los usuarios junto con los controladores específicos para que esto sea posible, de manera que convirtiendo estos nodos en autónomos del sistema en lo que respecta al plano de aplicación [139]. Con esta minimización, la información esencial proporcionada por el plano de control se almacena temporalmente en caché en la parte *edge* o incluso en nodos intermedios. De esta manera, esta información está disponible para los nodos *edge* en caso de problemas de conectividad de red en los que se pierde la conexión directa con el componente central, cumpliendo con una de las principales características de las aplicaciones *edge-native*, como se ha expuesto en el punto 2.3 del estado del arte de esta tesis.

KubeEdge se sitúa como la opción más destacada entre las soluciones analizadas de este tipo [139][140]. KubeEdge es un *framework* de código abierto basado en la arquitectura de Kubernetes con el propósito principal de trasladar las funcionalidades completas de K8s al *edge* [141]. Este proyecto se encuentra en continuo desarrollo bajo el paraguas de la CNCF, habiendo alcanzado el estado de *Graduated project* en octubre de 2024. La idea principal de esta tecnología se basa en mantener todo el plano de control en la parte *cloud*, donde hay una mayor disponibilidad de recursos de computación, y trasladar solamente el plano de

aplicación (es decir, los contenedores desplegados por los usuarios) a la parte *edge* para así dedicar todos los recursos computacionales a este propósito, ya que son limitados en esta capa del continuo. Además, la tarea de controlar la comunicación con los dispositivos *far-edge* sin capacidades de computación real (IoT, sensores, cámaras...) se asigna a los nodos más cercanos físicamente de ellos, lo cual hace posible que el consumo de memoria del EdgeCore (el componente *edge* de KubeEdge) sea de alrededor de 70 MB. La arquitectura de KubeEdge se divide en tres niveles:

- **Cloud:** para que KubeEdge funcione correctamente, es necesario contar con una distribución de K8s en funcionamiento en la parte *cloud* que interactúe con una instancia del CloudCore (el componente *cloud* de KubeEdge) desplegada en el mismo clúster. Además, este componente debe exponerse necesariamente mediante una IP pública, o que al menos sea alcanzable por los dispositivos *edge* que pretenda controlar. Esta parte *cloud* de KubeEdge incluye controladores para sincronizar el estado de todos los nodos *edge* y de los dispositivos IoT conectados a los nodos.
- **Edge:** en cada nodo *edge* se despliega el componente EdgeCore en forma de binario (no está virtualizada en forma de contenedor) directamente sobre la máquina. Este componente dirige la comunicación entre los contenedores de aplicaciones, los dispositivos conectados y el CloudCore. Los *Pods* de K8s (las cargas de trabajo que en realidad se orquestan y despliegan en un clúster KubeEdge) se ejecutan en este tipo de nodos, siendo su despliegue controlado por la parte *cloud*. La principal diferencia en comparación con una distribución ligera de K8s es que la parte de *edge* de KubeEdge no constituye un nodo de K8s en sí mismo, ya que prescinde de la API de K8s y de su plano de control, entre otros, debido a que sus funcionalidades son reemplazadas por el mismo EdgeCore, cuyos componentes internos se ilustran en la Figura 2.10, tomada del repositorio oficial de KubeEdge. Asimismo, estos nodos no necesitan tener una IP pública, sino que basta con que puedan alcanzar el *endpoint* del CloudCore, permitiendo que estén conectados a redes privadas o con más restricciones.

  Además, los nodos *edge* de un despliegue de KubeEdge pueden incluir un *broker* MQTT para interactuar con unos mapeadores de dispositivos desarrollados específicamente para asegurar su compatibilidad con esta tecnología (llamados *device mappers,* los tipos de mapeadores de protocolos de dispositivos disponibles son Bluetooth, Modbus y OPC-UA, pero además se ha puesto a disposición de los desarrolladores una biblioteca escrita en Go para crear mapeadores para otros protocolos, o incluso adaptar los ya existentes). Estos *mappers* se encargan de la interacción y control de los dispositivos, así como de la gestión de su

ciclo completo de funcionamiento (el concepto de *lilfecycle management,* LCM).

- **Dispositivos:** dispositivos típicos del ámbito del IoT, con capacidades de computación prácticamente nulas. Estos dispositivos interactúan con los nodos *edge* mediante los *mappers* anteriormente descritos utilizando varios protocolos industriales para el intercambio de datos.

El principal inconveniente de KubeEdge radica en la poca claridad de su documentación oficial para llevar a cabo una instalación completa en diferentes entornos, como puede atestiguar el doctorando debido a su experiencia en el uso de esta tecnología, que lo ha llevado a participar en los canales de reporte y debate (secciones de *issues* y *discussions*) de su repositorio de código de GitHub. No obstante, es necesario admitir que esta documentación ha ido mejorando a medida que ha avanzado la realización de esta tesis doctoral.

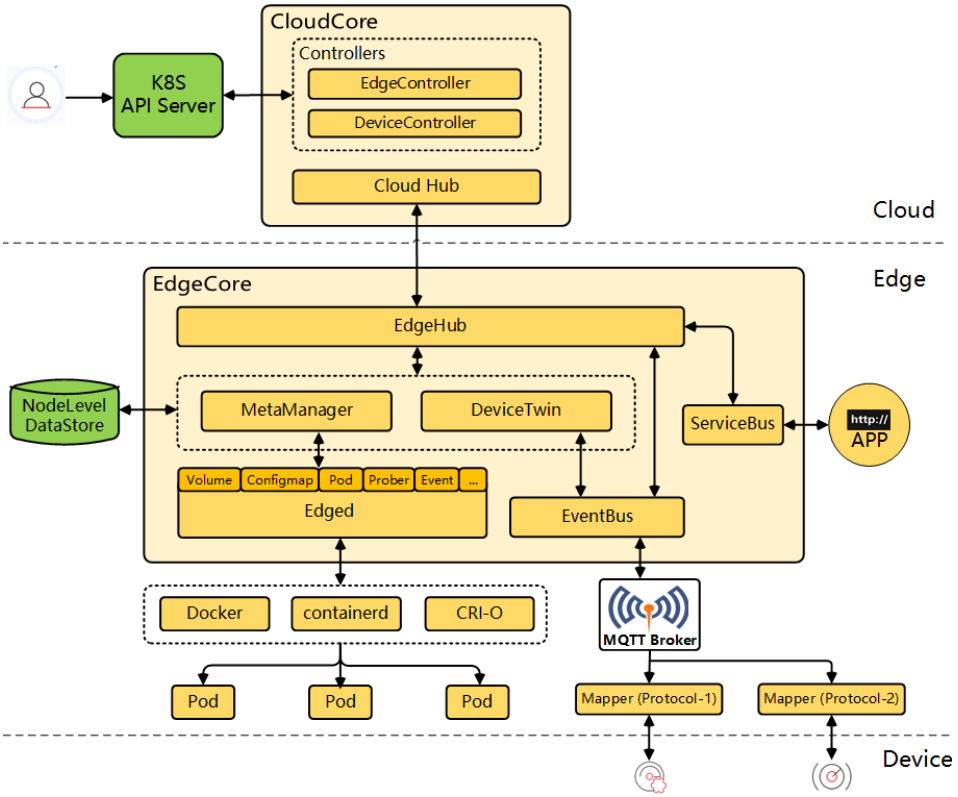


Figura 2.10: Arquitectura de KubeEdge [141]

Una característica destacable de KubeEdge que demuestra que ha sido desarrollado teniendo en cuenta patrones de diseño *edge-native* es su capacidad de gestionar conexiones de red deficientes entre sus nodos *cloud* y *edge*, de manera que la sincronización entre los componentes CloudCore y EdgeCore se realiza solo bajo condiciones de red estables, logrando así una gestión fluida del entorno. De esta manera soluciona un problema frecuente en entornos *Cloud-Edge* que no se tiene en cuenta en K8s debido a su naturaleza *cloud-native*, ya que este tipo de aspectos

no relacionados con las características de los entornos en la nube fueron desestimados en su diseño original. Igualmente, KubeEdge ha intentado adaptar una funcionalidad *cloud-native* de K8s a un entorno *edge* como lo es el *service mesh* mediante el desarrollo del componente EdgeMesh [142]. Con este enfoque, este orquestador habilita una comunicación transparente entre los servicios desplegados en entornos de red complejos, además de establecer escenarios de alta fiabilidad previniendo problemas de red potenciales mediante la distribución de metadatos de red a través de todos los agentes de EdgeMesh y la integración de servidores DNS ligeros.

Inicialmente, el EdgeMesh formaba parte del componente EdgeCore (ver Figura 2.10), pero debido a su complejidad, se decidió seguir su desarrollo como un componente independiente que puede instalarse en todos los nodos *edge* para permitir que los servicios desplegados en ellos se unan al *service mesh* del clúster [142]. Entrando en los detalles de implementación de EdgeMesh, el elemento central de este desarrollo es la librería descrita en el apartado previo 2.5.2: *libp2p*, por lo que EdgeMesh establece una red de comunicaciones P2P segura entre todos los nodos de un clúster de KubeEdge (tanto *cloud* como *edge*) utilizando una serie de nodos *relay* previamente configurados. Cuando el agente de EdgeMesh se instala en un nodo *edge*, el módulo *proxier* de este agente crea una serie de reglas en las *iptables* del nodo en cuestión para que le redirija todas las peticiones DNS de todos los *Pods* desplegados en él. Seguidamente, el EdgeCore comprueba en su componente *dns resolver* (que cuenta con un pequeño caché para reducir el número de peticiones al CoreDNS del clúster y evitar posibles problemas en la conexión) el *Pod* al que apunta este servicio y en qué nodo está desplegado realmente, para después enviar la petición original a su destino final a través del *stream* de *libp2p* previamente establecido con el nodo en cuestión. Por último, su arquitectura se ilustra en la Figura 2.11, que ha sido extraída del repositorio oficial de EdgeMesh.

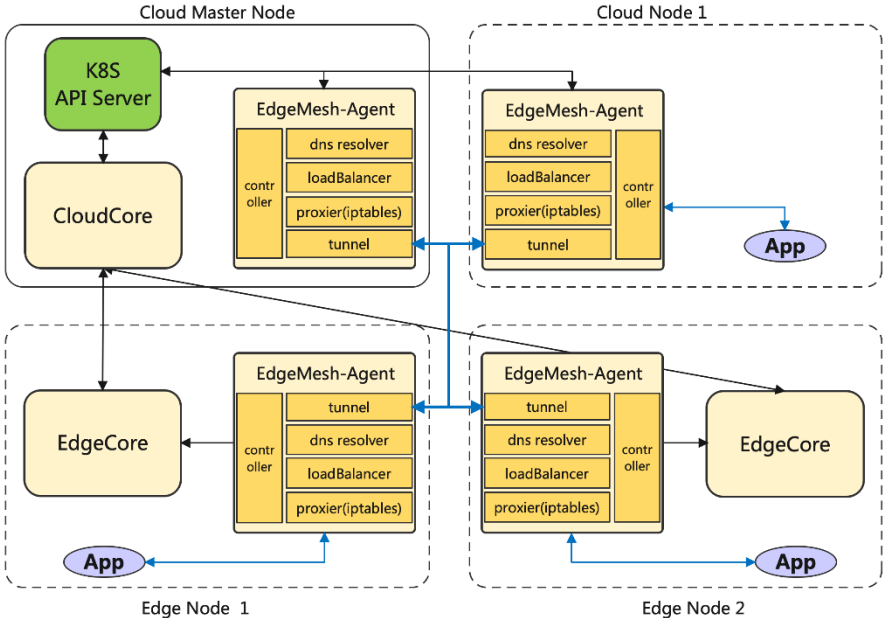


Figura 2.11: Arquitectura del EdgeMesh de KubeEdge [142]

En términos operativos, KubeEdge ha sido capaz de pasar con éxito una serie de pruebas de escalabilidad en un entorno laboratorio que demostraron que el *framework* era capaz de orquestar un millón de *Pods* desplegados en 100.000 nodos *edge*, cumpliendo con los Indicadores de Nivel de Servicio (SLI) y Objetivos de Nivel de Servicio (SLO) de K8s. En esta serie de pruebas se gestionó toda la infraestructura como un continuo de computación representado por un solo clúster de K8s [143]. De manera similar, es importante describir un caso de uso que puede ser ilustrativo para el uso en producción de KubeEdge. Este piloto fue presentado durante el *Kubernetes on Edge Day Europe 2021* por Huan Wei [144], y consistió en el despliegue de más de 100.000 dispositivos de monitorización a lo largo del puente Hong Kong-Zhuhai-Macao, uno de los puentes más largos del mundo. La parte *edge* de KubeEdge (su componente EdgeCore) se instaló en cada uno de los dispositivos ubicado a lo largo del puente, por lo que todos esos dispositivos fueron gestionados de forma centralizada por la parte *cloud* de KubeEdge (el CloudCore), que a su vez fue desplegada en un centro de datos de la nube pública. Cada dispositivo de monitorización era capaz de recopilar datos de 14 sensores diferentes (CO2, PM2.5, temperatura, humedad...) a través de su *mapper* específico, y los datos se procesaron localmente utilizando algoritmos de inferencia de IA desplegados como *Pods* en estos mismos nodos *edge*. De esta manera, solo fue necesario enviar a la parte *cloud* una pequeña parte de los datos previamente seleccionada e imprescindible a través de una conexión 5G estable, pero que, en caso de problemas de conectividad de red, un mecanismo de caché situado en el *edge* aseguraba que no se produjera una pérdida de datos durante el proceso gracias a los mecanismos *edge-native* proporcionados por KubeEdge. Aprovechando la sinergia *Cloud-Edge* lograda en el proyecto KubeEdge, la misma comunidad de desarrolladores intentó utilizar esta tecnología para mejorar la ejecución de servicios relacionados con modelos de inteligencia artificial (IA) en el continuo. Para este propósito, desarrollaron la herramienta Sedna, un proyecto enfocado en implementar capacidades de entrenamiento e inferencia colaborativas a lo largo del continuo de computación [145].

A pesar de que KubeEdge es la principal referencia en esta categoría de soluciones, es de una cierta lógica que no sea el único *framework* de este tipo que existe, ya que intenta solucionar un problema a la orden del día en el ámbito del *edge computing*. La CNCF aceptó como miembros en fase *Sandbox* a otros dos proyectos: OpenYurt [146] (el primer proyecto de código abierto llevado a cabo por el gigante chino Alibaba) y SuperEdge [147] (cuyo desarrollo es liderado por Tencent junto con Intel, VMware, Huya, Cambricon, Captialonline y Meituan). Tanto OpenYurt como SuperEdge están más enfocados a la implementación de centros de datos con formato reducido para ser desplegados en infraestructura del *edge* debido a sus mayores requisitos computacionales en comparación con KubeEdge. Una de las características más destacadas de OpenYurt es la integración con EdgeX Foundry para la gestión de dispositivos [148]. Ambos proyectos comparten el establecimiento de túneles seguros entre el plano de control (desplegado en la parte *cloud*) y los nodos *edge* como paso previo para poner en

práctica las comunicaciones bidireccionales necesarias para su correcto funcionamiento. De esta manera, estas tecnologías permiten superar los posibles obstáculos que puedan aparecer debido a la interacción entre redes heterogéneas a las que pueden pertenecer los nodos *edge*. Según la misma compañía Alibaba (recordemos, la impulsadora de OpenYurt), la principal diferencia entre OpenYurt y KubeEdge es que el primero es menos disruptivo con la arquitectura de K8s, ya que solo mejora K8s mediante el uso de *plugins* y Operadores, mientras que el segundo intenta rediseñar algunos componentes como *kubelet* o *kube-proxy*.

Otra tecnología relevante para trasladar la orquestación de contenedores al *edge computing* es Open Horizon (OH), desarrollada inicialmente por IBM y posteriormente donada a la Linux Foundation [149]. Esta tecnología comparte con KubeEdge la intención de mover las cargas de trabajo a la capa *edge* del continuo de computación, pero manteniendo el plano de control (gestión de servicios de usuario y nodos *edge*) en la nube o en un entorno centralizado. Además, OH promete gestionar hasta 10.000 dispositivos *edge* simultáneamente desde una única instancia de su Management Hub. La arquitectura de Open Horizon se divide en dos componentes principales:

- **Management Hub:** ubicado en la nube centralizada donde debe haber en funcionamiento un clúster de K8s. Se encarga de supervisar el plano de control en cuanto a los despliegues y los nodos y dispositivos *edge*.
- **Edge Agent:** este componente se divide a su vez en dos subtipos dependiendo del tipo de carga de trabajo que ejecutará el nodo. El Edge Device Agent está destinado a dispositivos con limitaciones de recursos capaces de ejecutar contenedores usando Docker o Podman como *container engine*, mientras que el Edge Cluster Agent está destinado para equipos en los que se puede instalar una distribución de K8s.

Open Horizon también está ganando popularidad entre la comunidad interesada en entornos *edge-native*, dado que los desarrolladores de EVE-OS (mencionado en el apartado 2.6.4, ambos son proyectos oficiales de la Linux Foundation) planean admitir cargas de trabajo basadas en OH, lo que supondría una gran mejora en la compatibilidad del proyecto, más teniendo en cuenta que los *Pods* de K8s ya son compatibles de forma nativa.

Finalmente, Baetyl se trata de otro proyecto que comparte algunos conceptos clave con KubeEdge y Open Horizon, ya que su arquitectura se divide en: (i) Cloud Management Suite y (ii) Edge Computing Framework [150]. Sin embargo, Baetyl solamente admite nodos *edge* con un mínimo de 1 GB de RAM que sean capaces de ejecutar una distribución de K8s (recomiendan el uso de K3s en entornos con recursos limitados), no siendo aún capaz de orquestar contenedores por sí mismo. Esta herramienta también está incluida en el nivel 1 de Linux Foundation Edge, pareciendo prometedora en un inicio, aunque, en el momento de la escritura de esta

tesis, se encuentra estancada en una etapa preliminar con una clara falta de documentación.

Otros tipos de soluciones

Existen algunos otros *frameworks* que difieren de la especificación de K8s pero que comparten el propósito de orquestar contenedores. Un ejemplo bastante representativo de estas alternativas es ioFog, desarrollado por la Eclipse Foundation. Este *framework* presenta algunas similitudes con Kubernetes, como el uso del comando *iofogctl* (que sería equivalente al comando *kubectl* de K8s) y la definición de microservicios mediante el lenguaje YAML. Además, tiene puntos en común con otras tecnologías específicas del *edge* mencionadas previamente, como contar con una arquitectura basada en el despliegue de un controlador en la nube que está a cargo de los agentes que se ejecutan en los nodos *edge*, en este caso llamado Edge Compute Network [151].

Para concluir esta subsección, es interesante describir el *framework* Akri, que también está bajo el paraguas de la CNCF, como un proyecto *Sandbox*. Akri se enfoca en el desarrollo de una "Interfaz de Recursos de Kubernetes" que permita exponer una variedad de dispositivos finales o IoT heterogéneos ubicados en el nivel más bajo del continuo como recursos en un clúster de K8s, por ejemplo, cámaras IP o dispositivos USB conectados a la misma máquina que ejerce como un nodo de un clúster de K8s [152]. A diferencia de las tecnologías vistas en esta subsección, Akri no se trata de distribución ligera de K8s, ni tampoco pretende orquestar contenedores en la parte *edge* de la arquitectura, sino que su propósito consiste en complementar nodos del *edge* permitiendo que los dispositivos finales sean reconocidos desde herramientas de gestión de más alto nivel (como K8s), proporcionando una capa de abstracción para los dispositivos de manera similar a como lo hace el CNI (Container Network Interface) de K8s para la red [64]. Esta es la principal diferencia entre Akri y las tecnologías anteriormente descritas (KubeEdge, Open Horizon...), y la razón por la que no está incluida en la Tabla 2.5. Sin embargo, es relevante considerarlo para mejorar la gestión de los dispositivos IoT en entornos del continuo *Cloud-Edge-IoT*.

Entrando en los detalles técnicos de su funcionamiento, los nodos interactúan con el servicio Akri Agent que se ejecuta en el nodo K8s más cercano del clúster, de manera que Akri extiende las funcionalidades de K8s, pero no lo adapta a escenarios *edge-native* en contraste con, por ejemplo, KubeEdge. Akri se basa en un conjunto de Discovery Handlers de dispositivos que se inspira en protocolos de gestión y comunicación para dispositivos industriales, tales como ONVIF, udev y OPC UA, así como la posibilidad de extender este conjunto con *handlers* personalizados [153]. Cuando un nuevo dispositivo es descubierto por los *handlers*, Akri crea un servicio K8s para monitorear su estado y proporcionar alta disponibilidad en caso de que un nodo pierda la conexión de red o se desconecte.

Tabla 2.5: Comparación de tecnologías para la orquestación de contenedores en el *edge*

Tecnología	Basada en K8s	*Edge Service Mesh*	Autonomía de los nodos *edge*	Arquitecturas CPU soportadas en el *edge*	Despliegues en entornos de producción	Responsable y estado actual	Popularidad
KubeEdge	Sí	Sí, completamente descentralizado usando EdgeMesh	Sí	x86_64, ARMv8, ARMv7	Puente Hong Kong–Zhuhai–Macao Centro de datos de China Mobile para el Internet industrial	CNCF Graduated	6.9k *stars* 1.7k *forks*
OpenYurt	Sí	Sí, usando Raven	Sí	x86_64, ARMv8, ARMv7	Hema Fresh's customer goods store AI project Video cloud migration in transportation	CNCF Sandbox	1.7k *stars* 298 *forks*
SuperEdge	Sí	Sí, usando ServiceGroups	Sí	x86_64, ARMv8	No documentado	CNCF Sandbox	407 *stars* 190 *forks*
Open Horizon	Sí	No	No	x86_64, ARMv8, ARMv7, ppc64le	IBM Edge Application Manager	LF Edge Stage 2	73 *stars* 99 *forks*
Baetyl	Sí	No	Temporal	x86_64, i386, ARMv8, ARMv7	No documentado	LF Edge Stage 1	1.9k *stars* 323 *forks*
Project Flotta	Sí	No	No	x86_64, ARMv8	No documentado	Red Hat Archivado	21 *stars* 21 *forks*
Eclipse ioFog	No	Sí	Sí	x86_64, ARMv8, ARMv7	No documentado	Eclipse Foundation	271 *stars* 33 *forks*

2.6.6. *Serverless*

El paradigma llamado *serverless* ("sin servidor") surgió para intentar abstraer a los usuarios de las características específicas del *hardware* subyacente de la infraestructura en el momento de desplegar sus servicios. Este concepto no significa que se prescinda realmente de un servidor o una máquina para el despliegue de aplicaciones, cosa que lógicamente sería imposible, sino que se elimina de manera conceptual el requerimiento de tener un servidor único o dedicado para un cierto tipo de servicios, aunque los servicios se sigan ejecutando en infraestructura o servidores reales. De esta manera, se ofrece a los usuarios finales la posibilidad de ejecutar sus aplicaciones sin tener que preocuparse por: (i) la infraestructura donde realmente se desplegará y (ii) durante cuánto tiempo reservarla.

Este paradigma ha tenido especial éxito en los servicios ofrecidos en la nube pública, ya que se basa en la suposición de que los recursos de cómputo disponibles están bien identificados (por ejemplo, en centros de datos) y que la infraestructura es fácilmente gestionable y accesible de manera uniforme por un solo operador (el proveedor de servicios *cloud*). Sin embargo, estos principios son exactamente opuestos a la realidad del *edge computing*, donde los responsables del desarrollo y ejecución del *software* no pueden abstraerse del *hardware* en el que realmente se ejecutará, dado que para el despliegue de una aplicación se debe tener en cuenta la heterogeneidad subyacente de los recursos. Como ejemplo comparativo, Google cuenta en sus centros de datos con miles de VMs con características similares, por lo que en términos de recursos y de operaciones no es importante en qué máquina se ejecute un servicio. Sin embargo, en un entorno *edge*, la ejecución de un servicio difiere ostensiblemente si se realiza en una Raspberry Pi o en un pequeño dispositivo industrial.

De igual manera que se ha visto en las secciones anteriores en el caso de la ejecución y orquestación de contenedores, han aparecido algunas iniciativas para aplicar mecanismos *serverless* en el *edge*. En este caso, cobran especial relevancia aquellas que se basan en el despliegue dinámico y temporal de los servicios para escalar o reducir bajo demanda los recursos destinados a su ejecución, optimizando el consumo de recursos (lo cual es de suma importancia en el *edge*). OpenFaaS [154] y Knative [155], que están integrados con Kubernetes, son los proyectos más relevantes que siguen este enfoque. Knative (en fase de incubación de la CNCF) también proporciona un motor completo basado en eventos definidos con el formato de CloudEvents, una especificación para estandarizar la definición de eventos bajo el paraguas de la CNCF [156], de manera que abre un amplio abanico de posibilidades en arquitecturas basadas en K8s en el *edge*. Con la inclusión de este motor basado en eventos, los servicios desplegados pueden emitir diferentes eventos con el propósito de activar algunas funcionalidades o cargas de trabajo sin modificar el código fuente de las aplicaciones desplegadas.

El paradigma *serverless* podría aprovecharse para permitir que equipos *edge* con menor capacidad de computación soliciten la ejecución de funciones a equipos más potentes que se sitúen en la capa de arquitectura directamente superior, por ejemplo, un análisis de sus métricas en tiempo real para saber monitorizar su rendimiento o detectar posibles ciberataques. Esta acción habilita una delegación efectiva y colaborativa de las responsabilidades de ejecución de servicios para balancear las capacidades de computación en entornos con recursos de computación limitados. Un ejemplo que ilustra esta arquitectura a la perfección es la integración de uno de estos controladores *serverless* (Knative o OpenFaaS) con el *broker* de mensajería *edge-native* NATS (descrito en el punto 2.4.2), que proporciona una integración nativa para el manejo de mensajes en formato CloudEvents, los cuales podrían utilizarse para desencadenar la ejecución de funciones *serverless* en entornos *edge*.

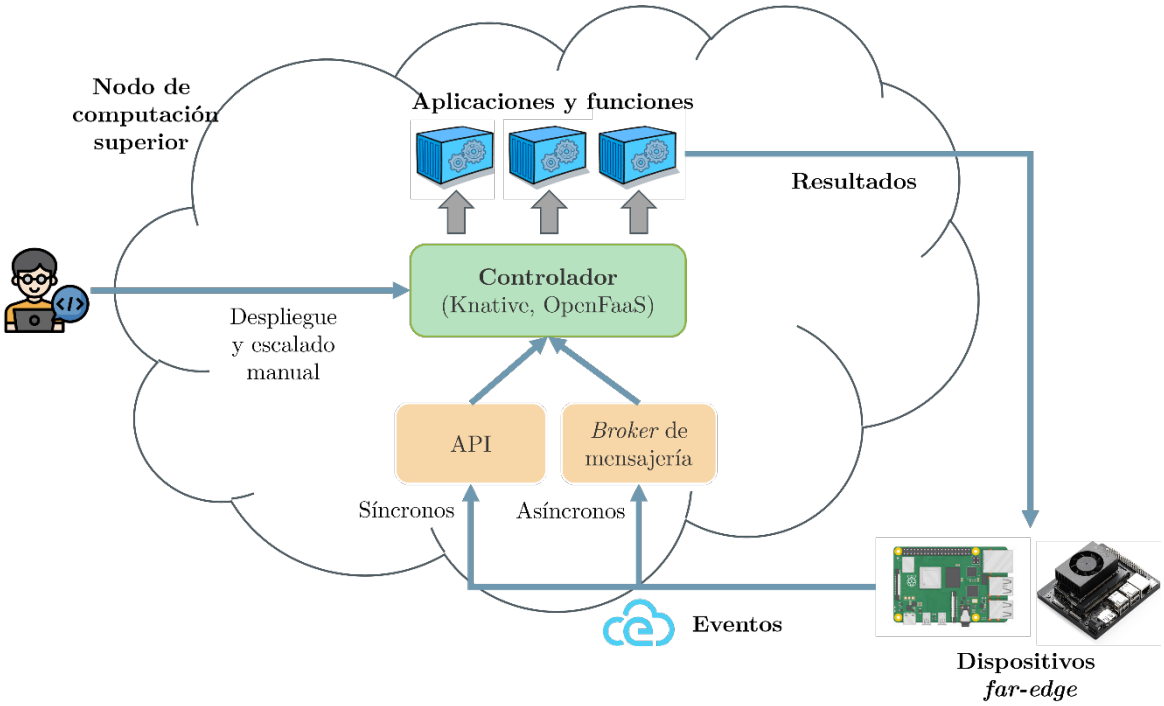


Figura 2.12: Ejecución de funciones *serverless* en el continuo de computación

2.6.7. Empaquetado de aplicaciones

En la sección 2.6.2 se ha descrito con detalle el proceso estandarizado por la OCI para empaquetar aplicaciones en imágenes de contenedores, las cuales posteriormente son ejecutadas como contenedores por los *container runtimes*. Por lo tanto, queda claro que el primer paso para lograr virtualizar una aplicación en formato de contenedor es construir una imagen a partir de su código fuente. La especificación de imágenes mediante Dockerfiles se ha convertido en el estándar *de facto* gracias a que forma parte del ecosistema de Docker, aunque existen otras alternativas para construir imágenes como Buildah [157] (forma parte del

ecosistema de youki y crun e incluso es compatible con estas Dockerfiles) o Cloud Native Buildpacks [158] de la CNCF. Este hecho también se debe a que para crear imágenes de manera manual (sin el uso de Dockerfiles o herramientas facilitadoras) se necesitan conocimientos muy específicos sobre la materia, además de no ser nada práctico cuando se añaden a este proceso herramientas de automatización y DevSecOps como *pipelines* de CI/CD. En resumen, en estos ficheros Dockerfile se describen las capas que conforman una imagen mediante el uso de palabras reservadas (*FROM*, *RUN, CMD, ENTRYPOINT, COPY*...) que ejecutan acciones determinadas, como definir el uso de una imagen base, copiar ficheros del código fuente o ejecutar ciertos comandos. En última instancia, cuando el Dockerfile está listo, se deben utilizar las debidas herramientas para construir automáticamente la imagen, que en el caso de Docker sería el uso comando *docker build*.

Después de ser creadas, las imágenes de contenedores son posteriormente publicadas en un registro o repositorio de imágenes para que puedan ser descargadas y ejecutadas por los *runtimes*. Actualmente existen numerosas alternativas de repositorios de imágenes públicos, siendo el más utilizado Docker Hub [159] por motivos evidentes: no solo está gestionado por la misma compañía Docker, sino que los *container runtimes* buscan las imágenes por defecto en Docker Hub (incluso en K8s) si no se especifica claramente que la imagen pertenece a otro repositorio. Asimismo, repositorios de código como GitHub o GitLab han incluido registros de imágenes en su ecosistema. No obstante, se han desarrollado tecnologías bajo el paraguas de la CNCF como Harbor [160] (repositorio tradicional centrado en la seguridad de las imágenes, que ya ha alcanzado el estado de *Graduated project*) o Dragonfly [161] (más innovador dado que está basado en comunicaciones P2P).

Llegados a este punto, aún no se ha alcanzado el nivel deseado de empaquetado, ya que el despliegue de aplicaciones avanzadas no se realiza solamente con la ejecución de contenedores a partir de imágenes. También es necesario especificar requerimientos adicionales del servicio como exposición de puertos de red, necesidades de almacenamiento y persistencia o dependencias con otros servicios. Docker fue igualmente uno de los pioneros en la creación de herramientas más avanzadas como Docker Compose, que permite ejecutar diversos contenedores simultáneamente con un orden predefinido mediante la definición de estos requerimientos en un único fichero de configuración en formato YAML [162].

Sin embargo, con la llegada de Kubernetes esta especificación pasó a estar obsoleta, puesto que este orquestador de contenedores añade una capa más de complejidad al añadir elementos de configuración propios en la ecuación, como *Services* o *Pods*. K8s heredó de Compose el uso de YAML para definir sus propios despliegues en ficheros llamados manifiestos, pero dada la anteriormente comentada complejidad, el despliegue de una simple aplicación se compone de varios manifiestos que pueden ser definidos en un mismo fichero o en un conjunto de ellos. Por este motivo, empezaron a surgir herramientas para automatizar el despliegue de estos manifiestos y avanzar hacia el empaquetado uniforme de aplicaciones complejas, como lo son Kustomize y Helm. Mientras Kustomize es una herramienta

más simple que actúa directamente sobre los manifiestos para sustituir diferentes partes de ellos por contenidos predefinidos por los usuarios [163], Helm es una herramienta más compleja basada en la técnica de *templating*, que consiste en la generación de contenido dinámico a partir de una estructura definida (en el caso de Helm de unos ficheros previamente creados) [164]. Los usuarios deben definir un Helm chart, que se compone por *templates* de manifiestos de K8s en los que configuran ciertas partes de estos para ser sustituidas por valores definidos por los usuarios en archivos llamados *values*. De igual manera que existen repositorios de imágenes, también existen repositorios públicos de Helm charts, siendo Artifact Hub el repositorio público más utilizado [165], igual que está nativamente integrado tanto en GitHub como en GitLab.

Por último, aunque los Helm charts son una herramienta muy interesante para el empaquetado, configuración y despliegue de aplicaciones, su trabajo termina en el momento en el que una aplicación es desplegada, ya que carecen de la capacidad para gestionar dinámicamente el ciclo de vida de las aplicaciones o las también conocidas como operaciones "del segundo día o de día 2" (*Day-2 operations*). Es en este contexto donde el uso de los Operadores de K8s son de gran interés, porque amplían la funcionalidad proporcionada por Kubernetes al automatizar tareas operativas complejas, como lo son la monitorización, escalado, realización de copias de seguridad, capacidades de autorrecuperación, a través de controladores personalizados. Estos controladores se encargan de que el estado de las aplicaciones coincida con el estado deseado definido por el usuario mediante la ejecución de funciones llamadas de reconciliación, que ejecutan una lógica totalmente personalizada para la aplicación [166]. Por ende, estas funcionalidades los hacen ideales para gestionar aplicaciones complejas y con estado que requieren mantenimiento y optimización continuos.

2.7. Alternativas a los contenedores

Como se ha ido explicando en los apartados anteriores del estado del arte tecnológico, los sistemas que siguen el patrón de diseño *cloud-native* giran en torno a la orquestación de cargas de trabajo basadas en contenedores, ya que indudablemente se trata del estándar *de facto* actual. Sin embargo, se puede afirmar que los contenedores no pueden establecerse como un enfoque final y definitivo en un campo tecnológico donde todo está en continuo cambio. De hecho, es bien sabido que los contenedores se encuentran lejos de ser una solución perfecta, aunque se hayan establecido como una tecnología dominante, puesto que presentan algunas debilidades (por ejemplo, en seguridad [167]), además de tener una capacidad de mejora limitada debido al alto grado de madurez con el que cuenta esta tecnología.

Respecto al uso de contenedores en escenarios relacionados con el *edge computing*, aunque al igual que en el entorno *cloud* están siendo ampliamente utilizados (como se puede ver en los diferentes apartados de la anterior sección 2.6), todavía cuentan con ciertas desventajas como pueden ser los requerimientos de

ejecución de sus *runtimes*, su alto consumo de memoria e incluso en el tamaño de las imágenes de contenedores. Por lo tanto, en los últimos años han surgido otras técnicas de virtualización que podrían competir con los contenedores, incluso algunas de ellas cuentan con una tendencia pujante en el campo de investigación estudiado en esta tesis. Sin embargo, esto no significa necesariamente que estas tecnologías vayan a sustituir a los contenedores a corto plazo, sino que simplemente pueden verse como un potencial complemento a los contenedores, pudiendo utilizarse para casos de uso específicos en los que encajen los requerimientos de las aplicaciones que se necesite virtualizar [168]. Es necesario recordar que este último es un aspecto muy importante, debido al hecho de que en última instancia se trata de ejecutar aplicaciones en un continuo de computación con unos requerimientos de funcionamiento específicos, que a su vez difieren de las demás aplicaciones que conforman cualquier ecosistema tecnológico.

2.7.1. MicroVMs y unikernels

En el apartado 2.6.1 se han enumerado brevemente las ventajas de los contenedores respecto con las VMs. Si bien estos argumentos son ciertos, las capacidades de aislamiento de las VMs (debido a que no comparten el mismo *kernel* entre instancias virtualizadas y usan un hipervisor) son interesantes para el *edge computing*, puesto que se trata de un entorno donde es necesaria una capa de seguridad adicional ya que los equipos se sitúan más cerca de las fuentes de datos y son potencialmente más vulnerables o propensos a ataques (incluso físicos). De este modo, la intención de aislar las aplicaciones a un nivel aún más bajo puede traducirse en una mejora significativa en la seguridad de las cargas de trabajo desplegadas por los usuarios. A diferencia de lo que ocurre con los contenedores, este enfoque elimina los entornos multiusuario (*multi-tenant*) no confiables, reduciendo así los riesgos asociados a la compartición de la infraestructura de computación entre servicios pertenecientes a diferentes propietarios o responsables.

A partir de estos hallazgos, ha surgido un nuevo enfoque para la virtualización de cargas de trabajo, que logra mantener la capacidad de aislamiento mientras omite algunos inconvenientes de las VMs. Este enfoque ha podido implementarse mediante la reducción del tamaño y los requisitos de las VMs, dando lugar a las llamadas microVMs o VMs ligeras, que pretenden ser incluso más ligeras y rápidas que los contenedores. No obstante, hasta la fecha no existen resultados concluyentes que lo demuestren de forma fiable. Algunos estudios [169] han comparado el rendimiento de los *runtimes* de contenedores (runC) y microVM (Kata Container), mostrando un mejor rendimiento del primero. Se argumentó que este bajo rendimiento podría deberse a los mayores tiempos de arranque de las microVMs (3.7 s en promedio), en comparación con los contenedores (0.633 s en promedio). La tecnología de microVM utilizada para la comparación (Kata Containers) es una herramienta de OpenStack cuyo objetivo principal es ejecutar VMs más ligeras en lugar de contenedores, manteniendo la compatibilidad con las especificaciones de

runtime e imagen OCI (revisar el punto 2.6.1). Al ser compatible con la especificación OCI, Kata permite que las imágenes Docker previamente construidas a partir de Dockerfiles se desplieguen como microVMs (contenedores Kata) [170]. Asimismo, este bajo rendimiento se ha intentado solucionar mediante el desarrollo de un *runtime* de contenedores más ligero orientado a la ejecución de cargas de trabajo basadas en microVMs, pero que a su vez es compatible con OCI: *RunD*, especialmente enfocado en resolver el problema de cuello de botella que se puede observar en ciertos despliegues de funciones *serverless*. RunD ha sido adoptado por Alibaba como su *runtime* de contenedores *serverless*, siendo capaz de ejecutar más de un millón de funciones y casi 4 mil millones de invocaciones diarias. Además, RunD ha demostrado que puede lanzar 200 VMs ligeras en un segundo y desplegar exitosamente 2500 de ellas en una máquina con 384 GB de memoria [171].

Otro enfoque para reemplazar o complementar los contenedores con microVMs son los unikernels. Según [172], "los unikernels son aplicaciones de propósito único que se especializan en tiempo de compilación en *kernels* independientes y se sellan contra modificaciones cuando se despliegan en una plataforma *cloud*". La idea principal detrás de los unikernels es usar solo la parte estrictamente necesaria del *user namespace* y del *kernel* de un sistema operativo, para de esta manera obtener un OS personalizado que se ejecutará en un hipervisor de tipo 1. Este OS personalizado se logra mediante el uso de un sistema operativo como si se tratara de una librería (*library OS* o libOS), eliminando la necesidad de un OS adicional completo [173]. Esto se traduce en una reducción del tamaño de la imagen y su tiempo de arranque, así como de su consumo de memoria y posible superficie de ataque. Sin embargo, esta tecnología de virtualización presenta a su vez muchas desventajas, entre las que destaca la falta de estandarización en comparación con los contenedores, además de la necesidad de reconstruir completamente un libOS para cada nueva aplicación con cualquier cambio mínimo, junto con la limitación de las capacidades de depuración y monitorización [174]. Las aplicaciones de Unikernel también son específicas para cada lenguaje de programación (los sistemas de desarrollo de Unikernel están disponibles solo para unos pocos lenguajes), lo que también es una limitación considerable para su adopción por parte de los desarrolladores de forma masiva. Por ejemplo, MirageOS es un libOS que permite construir unikernels usando el lenguaje OCaml junto con librerías que proporcionan soporte de redes, almacenamiento y concurrencia [175].

Centrando el foco en las tecnologías para el desarrollo y ejecución de unikernels, Nabla containers es un proyecto de investigación de IBM focalizado en construir una plataforma para manejar unikernels (por ejemplo, cargas de trabajo desarrolladas usando MirageOS) mediante el uso de su propio *runtime* de contenedores de bajo nivel, *runnc* [176][177]. Aunque runnc es *compatible* con el estándar OCI, la especificación de imagen de Nabla no lo es, por lo que este proyecto no admite aplicaciones empaquetadas siguiendo especificaciones de imagen de contenedor distintas a la de Nabla. Desafortunadamente, el proyecto Nabla se archivó oficialmente en mayo de 2023, aunque esto no significa que no hayan

surgido nuevos enfoques con el objetivo de convertir a los unikernels en una opción razonable con mejor integración en el entorno *cloud-native*. Es aquí donde se debe poner en valor el proyecto Unikraft, que se define como un "sistema automatizado basado en Linux para construir unikernels", se encuentra bajo el paraguas de la Linux Foundation (dentro de su Proyecto Xen) y fue financiado parcialmente por el proyecto H2020 UNICORE [178]. Aunque en su día existieron otras iniciativas similares, como Lupine, Unikraft presenta un rendimiento mejorado en comparación con sus alternativas y otras tecnologías de virtualización, como Docker (en cuanto a tiempo de arranque, tamaño de imagen y consumo de memoria). Unikraft utiliza un *runtime* compatible con OCI (*runu*) junto con libvirt (un kit de herramientas para interactuar con el hipervisor) para ejecutar unikernels. Su gran innovación es que runu admite de forma nativa la ejecución de cargas de trabajo previamente empaquetadas según la Especificación de Imagen OCI (a diferencia de Nabla), permitiendo la interacción de unikernels construidos con Unikraft con contenedores y plataformas nativas de la nube como K8s [179]. Para facilitar el uso y adopción de esta herramienta, sus desarrolladores lanzaron KrakfKit, también de código abierto, que ofrece una serie de herramientas y un marco de desarrollo basado en Go para construir unikernels [180]. Sin embargo, este prometedor enfoque de unikernel no ha tenido la adopción esperada en entornos de producción, quedándose circunscrita al ámbito académico por el momento [181].

Finalmente, y a modo de resumen del contenido explorado en este subapartado, en la siguiente figura se establece una comparación gráfica entre los bloques de arquitectura de las tres tecnologías investigadas: microVMs, unikernels y contenedores.

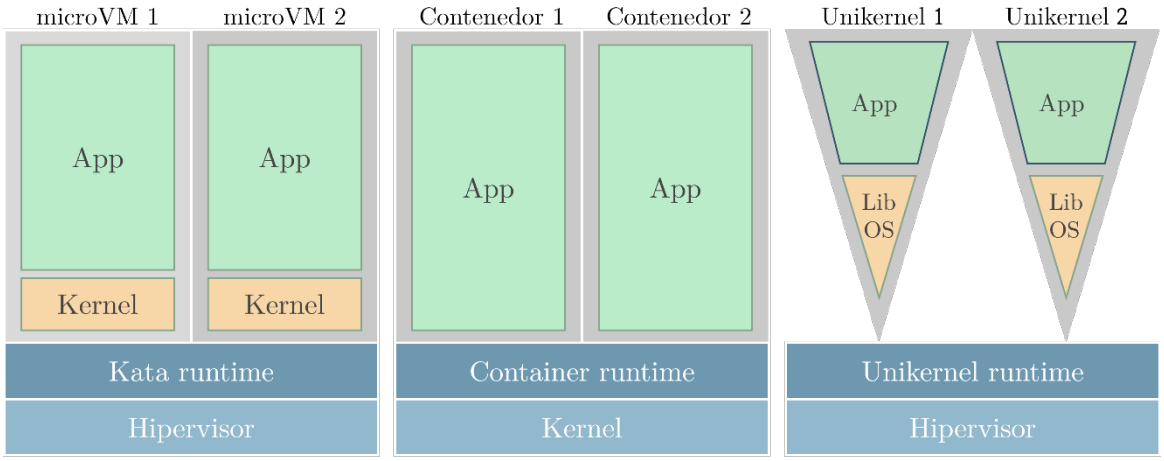


Figura 2.13: Comparación de diferentes técnicas de virtualización: microVMs o VMs ligeras, contenedores y unikernels

2.7.2. WebAssembly

Se puede afirmar debido a la expectación que ha generado en el mundo de las TIC, y especialmente en comunidades de desarrollo de *software* y de DevOps, que la tendencia más reciente y prometedora en la virtualización de aplicaciones es WebAssembly (Wasm). Esto se puede comprobar en el crecimiento de popularidad de sus repositorios de código, en el número de publicaciones relacionadas con esta tecnología o incluso en su inclusión en proyectos de investigación. De hecho, el profundo interés por esta tecnología es tal que el cofundador de Docker, Solomon Hiles, afirmó en 2019 en una famosa publicación en la red social Twitter (ahora llamada *X*) que, si Wasm junto con WASI hubieran existido en 2008, no habrían necesitado crear Docker, además que, según su punto de vista, Wasm ejecutado en el lado del servidor es el futuro de la computación [182]. Sin embargo, el año en el que se realizó esta afirmación también podría verse como el alto nivel de complejidad que comprende la implementación real de esta tecnología, así como la dificultad que conlleva lanzar una tecnología que pueda establecerse como alternativa a un tipo de virtualización ampliamente establecido como los contenedores.

Centrando el foco en el aspecto técnico de esta tecnología, WebAssembly se define como "un formato de instrucción binario para máquinas virtuales basadas en pila (*stack-based* VMs)" desarrollado como un estándar abierto por el World Wide Web Consortium (W3C), que busca establecerse como una fuerte alternativa a los contenedores. Wasm permite que el *software* escrito en varios lenguajes de alto nivel (C++, Go, Rust, Java...) se compile y ejecute alcanzando un rendimiento casi nativo en aplicaciones web diseñadas para ejecutarse en navegadores web como Google Chrome o Mozilla Firefox [183][184]. Sin embargo, algunos estudios han demostrado que Wasm ofrece un rendimiento inferior en comparación con el código nativo, con un factor de 1.45 para Firefox y 1.55 para Chrome [185], después de ejecutar diferentes pruebas siguiendo el estándar de comparación (o de *benchmarking*) SPEC (Standard Performance Evaluation Corporation). Esta tecnología fue diseñada inicialmente para mejorar el rendimiento de aplicaciones exigentes que no ofrecen un rendimiento aceptable cuando se desarrollan en JavaScript con el propósito de adaptarlas al marco web (los navegadores pueden ejecutar código de JavaScript de manera nativa), en lugar de en lenguajes específicamente adaptados a su propósito de trabajo. A nivel general, el funcionamiento de Wasm es simple: las funciones de código nativo se compilan en módulos Wasm que luego son cargados y utilizados por una aplicación web (en código JavaScript) de una manera similar a cómo se gestionan los módulos ES6. En relación con estos módulos precompilados, en el artículo [186] se ha investigado la posibilidad de usarlos como una nueva técnica de ofuscación para insertar malware en aplicaciones web, lo que dificulta su detección por parte de los analizadores de malware. Este inconveniente puede verse como un claro ejemplo de los nuevos riesgos que siempre llevan asociadas las tecnologías innovadoras.

No obstante, el aspecto innovador y de interés para esta tesis doctoral que proporciona Wasm, no está relacionado con su funcionalidad original de ejecutar aplicaciones en los navegadores, sino con la posibilidad de trasladar la ejecución de estos módulos Wasm fuera de éstos, o, explicado de manera más concisa, mover su ejecución desde el lado del cliente (navegador web que se ejecuta en las máquinas del cliente) hacia el lado del servidor (en la misma máquina o infraestructura como si se tratara de una aplicación al uso). El tamaño reducido de los binarios de los módulos Wasm, junto con su bajo consumo de memoria, su gran aislamiento, su rápida capacidad de arranque (arrancan hasta 100 veces más rápido que los contenedores) y sus reducidos tiempos de respuesta, hacen de Wasm una tecnología muy prometedora para ejecutar cargas de trabajo en dispositivos *edge* e IoT [187]. De esta manera, la adopción de Wasm en entornos *edge* (donde los nodos con recursos limitados tienen dificultades para ejecutar contenedores) podría habilitar el incremento del número de servicios que se pueden ejecutar de manera simultánea en comparación con su ejecución en forma de contenedores [188][189].

Como se ha descrito previamente, el uso de Wasm permite el desarrollo de aplicaciones en cualquier lenguaje de programación, ya que existen compiladores que ofrecen compilación nativa a Wasm para la mayoría de ellos, por lo que los módulos Wasm son independientes del lenguaje en el que han sido programados. Estos módulos Wasm también son independientes de la arquitectura de la CPU de la máquina en los que se ejecutan, a diferencia de lo que ocurre en los contenedores, puesto que las arquitecturas de CPU se deben tener en cuenta en el momento de compilación de las imágenes de contenedores (es necesaria la creación de una imagen específica para cada arquitectura de CPU). Este doble nivel de independencia, tanto de lenguaje de programación como de arquitectura de CPU, dota a Wasm una gran versatilidad, proporcionándole de una clara ventaja adicional para lidiar con la heterogeneidad de arquitecturas presentes en el continuo *Cloud-Edge-IoT* [187].

Respecto al funcionamiento de WebAssembly, es necesario destacar que cuando se ejecuta un binario Wasm en un navegador web, éste usa las API web proporcionadas por el navegador para interactuar con los componentes externos necesarios a través de las llamadas al sistema adecuadas. Sin embargo, cuando los desarrolladores pioneros en esta tecnología comenzaron a explorar la ejecución de estos módulos fuera de los navegadores, el primer desafío con el que se encontraron fue que no existía una API equivalente. Por lo tanto, era prioritario llevar a cabo la estandarización, el diseño y el desarrollo un *software* "puente" para habilitar la interacción entre un módulo Wasm y las llamadas al *kernel* del sistema operativo anfitrión. Esta acción se realizó manteniendo dos principios fundamentales de Wasm: (i) la portabilidad (poder ejecutar el mismo código en diferentes sistemas) y (ii) la seguridad. Finalmente, este esfuerzo combinado resultó en la creación de la Interfaz del Sistema WebAssembly (WASI), una "interfaz de sistema modular para WebAssembly" con el propósito principal de habilitar la ejecución de Wasm en el lado del servidor mediante la creación y estandarización de diferentes APIs. Debido a la importancia de la seguridad en este entorno, WASI se diseñó

meticulosamente para minimizar la superficie de ataque de los módulos y así evitar que módulos Wasm malintencionados comprometan el sistema de la máquina anfitriona. Asimismo, WASI ha sido diseñado para ser independiente del *runtime* Wasm utilizado, el cual es el componente encargado de ejecutar módulos o aplicaciones Wasm, que puede ser comparado con los *runtimes* de contenedores de bajo nivel en cuanto a su funcionalidad, debido a que ambos son los componentes que realmente ejecutan las cargas de trabajo [190][191]. De esta manera, los módulos Wasm realizan llamadas WASI a un *runtime* de Wasm que seguidamente se traducen en las llamadas al sistema apropiadas para finalmente interactuar con el *kernel* del OS [192], como se muestra en el esquema de la Figura 2.14.

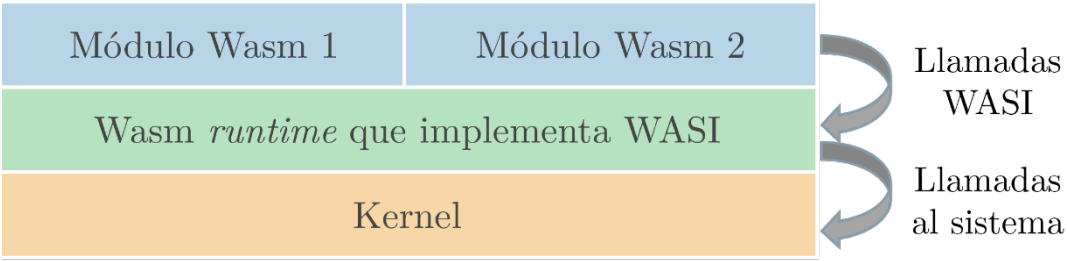


Figura 2.14: Arquitectura para la ejecución de cargas de trabajo de WebAssembly en el servidor

2.7.2.1. Estandarización: WASI 0.2 y el Wasm Component Model

Como se ha mencionado con anterioridad, WASI es una tecnología reciente, por lo que actualmente aún se encuentra en una fase de estandarización, la cual está dirigida y promovida por un subgrupo de la WebAssembly Community Group del W3C: el WASI Subgroup [193]. Este proceso de estandarización de WASI está organizado en una serie de propuestas de estandarización de funcionalidades expresadas mediante APIs (HTTP, sistema de archivos, aprendizaje automático...), que a su vez se dividen en 5 fases de desarrollo según el grado de madurez alcanzado [194]:

- **Fase 0 - Propuestas preliminares:** aún no se puede hablar de ellas como propuestas oficiales de WASI, su finalidad es más bien compartir ideas preliminares sobre la que construir propuestas más potentes. Estas propuestas las realizan contribuidores individuales sin el soporte del subgrupo WASI.
- **Fase 1 - Propuesta de nueva funcionalidad o API:** en esta fase la propuesta está aceptada por el subgrupo y se incluye en la lista de propuestas oficiales.
- **Fase 2 - Descripción de la funcionalidad disponible:** durante esta fase se crean diversos prototipos de implementación y un conjunto de tests.

- **Fase 3 - Fase de implementación:** se realizan lanzamientos de las implementaciones elegidas para dicha funcionalidad siguiendo el estándar SemVer (Semantic Versioning) [195].
- **Fase 4 - Estandarización:** la implementación de la funcionalidad se entrega al subgrupo de trabajo de WASI, el cual puede realizar pequeñas modificaciones.
- **Fase 5 - Funcionalidad completamente estandarizada:** una vez que el subgrupo de trabajo alcanza un consenso sobre la completa implementación de la funcionalidad, se proceden a realizar los trabajos necesarios de edición de la documentación y de publicación del código en la rama principal del repositorio de la propuesta.

El pasado 25 de enero de 2024 supuso un hito para el desarrollo de WASI, ya que el subgrupo WASI acordó el lanzamiento de WASI 0.2 (también conocido como WASI Preview 2) 5 años después del lanzamiento de la primera versión de WASI (la versión 0.1), por lo que las funcionalidades incluidas en este lanzamiento se pueden considerar estables (todas ellas se encuentran en fase 3) [196]. Además, otra novedad muy importante que conlleva asociada este lanzamiento es que a partir de ese momento WASI está oficialmente basado en el Wasm Component Model , por lo que sus propuestas de estandarización de funcionalidades pasan a estar definidas usando el lenguaje WIT (Wasm Interface Type) [197] además de ser implementadas en forma de componentes. En resumen, tomando las palabras escritas por Dan Gohman en el anuncio oficial de este lanzamiento: "WASI 0.2 es una API, pero también representa una nueva manera de definir APIs para Wasm" [196].

Antes de seguir con la descripción de WASI 0.2, es importante detenerse en el Wasm Component Model. Esta especificación, que es una extensión de la especificación matriz (*core*) de WebAssembly, comenzó a desarrollarse debido a la no existencia de un modelo estándar para la combinación de diferentes módulos Wasm en el momento su ejecución (de esta manera pueden usarse como si se tratasen de librerías en el desarrollo de código) ni para la alineación de tipos o estructuras de datos (por ejemplo, *strings* o listas) entre módulos. A nivel práctico, esto significa que los módulos se tenían que enlazar estáticamente en el momento de su compilación en lugar de hacerlo dinámicamente en tiempo de ejecución, además de no poder comunicarse efectivamente entre ellos mediante un procedimiento estándar y fácilmente portable. Explicado de manera más concisa, la implementación de los tipos básicos de datos difiere en los múltiples lenguajes de programación existentes (un *string* no se representa de igual manera en Rust que en Go, por ejemplo), por lo que la interacción entre módulos Wasm originalmente desarrollados utilizando diferentes lenguajes resultaba dificultosa (solamente podían exponerse funciones que devolvieran tipos básicos como enteros o números de coma flotante — *floats*) o directamente inalcanzable (sobre todo en tipos más complejos como listas o estructuras).

Como solución a esta problemática, en el Component Model se establece un formato común para la definición de tipos en los componentes mediante el uso del lenguaje WIT que a su vez está soportado por una ABI canónica (Cannonical ABI) cuya función es traducir estos tipos a código binario [198]. De esta manera, cada componente define los tipos y funciones que importa y exporta (sus *imports* y *exports*) y dependencias en las llamadas *interfaces* (definen como han de interactuar los componentes entre ellos, consolidando la capacidad de aislamiento de Wasm), que a su vez se engloban en los llamados *worlds*. De igual manera, estos se engloban en paquetes WIT (realmente son colecciones de interfaces y *worlds* definidas en varios archivos WIT) que pueden ser publicados en registros, siendo un elemento clave para el intercambio de tipos y definiciones en un ecosistema basado en componentes Wasm. Gracias a esta interfaz de interoperabilidad común proporcionada por el Component Model, finalmente se consigue que los componentes Wasm sean portables entre lenguajes de programación actuando como si se tratasen de librerías escritas en el mismo lenguaje (como por ejemplo la librería *fmt* de Go o *NumPy* de Python), sumando este nuevo tipo de portabilidad a las ya alcanzadas de arquitectura de CPU y de sistema operativo, permitiendo abrir un gran abanico de posibilidades a Wasm en un entorno heterogéneo como el continuo *Cloud-Edge-IoT*. A modo de ejemplo ilustrativo, si un servicio tiene una dependencia especificada en una interfaz WIT (importa esta interfaz), puede cambiar dinámicamente el componente Wasm que esté usando como dependencia, siempre y cuando este último exporte la interfaz WIT que el componente primario necesita importar. En este caso, la interfaz WIT estaría definida como un estándar, pero su implementación variaría a nivel de código fuente.

```
1  package thesis:calculator
2
3  interface calculator {
4    add: func(a: float32, b: float32) -> float32;
5    subtract: func(a: float32, b: float32) -> float32;
6    multiply: func(a: float32, b: float32) -> float32;
7    divide: func(a: float32, b: float32) -> result<float32, string>;
8  }
9
10 world calculator {
11   import host: interface {
12     log: func(msg: string);
13   }
14   export calculator
15 }
16
```

Figura 2.15: Ejemplo de la definición de un componente Wasm en formato WIT

Después del punto de ruptura que ha supuesto la aparición de WASI 0.2 junto con el anteriormente descrito Component Model, depende del desarrollador elegir la forma en la que pretende compilar su programa (desde el código fuente) al binario WASM:

- Compilación de acorde a la especificación general de WebAssembly, o, dicho de otro modo, compatible con WASI 0.1 (antes de la aparición del Component Model). El resultado obtenido es un módulo Wasm (llamados *core Wasm modules*). Esta opción resta versatilidad, pero su soporte está más extendido en Wasm *runtimes* y lenguajes de programación.
- Compilación siguiendo los estándares definidos por el Component Model, por lo que sería compatible con las ventajas que aporta WASI 0.2. El resultado obtenido es un componente Wasm. Sin embargo, aún se encuentra en una fase temprana de adopción, por lo que su compatibilidad y estabilidad es menor.

Volviendo a la estandarización de WASI, actualmente, las propuestas más maduras están en la fase 3, las cuales coinciden con las que se incluyeron en la versión 0.2.0, lo que se puede ver como otro indicador de la novedad de esta tecnología y del trabajo pendiente para que esta tecnología pueda utilizarse globalmente en entornos de producción reales, quedando de momento más restringida al ámbito de la investigación. Sin embargo, los avances que se han producido en este último año han permitido seguir mejorando la usabilidad de WASI. A modo de resumen, en la siguiente tabla se listan parte de las propuestas actuales de WASI que se encuentran en fases 2 y 3, junto con algunas propuestas interesantes en fase 1 [194]:

Tabla 2.6: Propuestas actuales de WASI

Propuesta WASI	Autor	Fase
I/O	Dan Gohman	3
Clocks	Dan Gohman	3
Random	Dan Gohman	3
Filesystem	Dan Gohman	3
Sockets	Dave Bakker	3
CLI	Dan Gohman	3
HTTP	Piotr Sikora et al.	3
I2C	Friedrich Vandenberghe et al.	2
Key-value Store	Jiaxiao Zhou et al.	2
Machine Learning (wasi-nn)	Andrew Brown y Mingqiu Sun	2
Runtime Config	Jiaxiao Zhou et al.	2
GFX	Mendy Berger y Sean Isom	2
Messaging	Jiaxiao Zhou et al.	2
Logging	Dan Gohman	1
SQL	Jiaxiao Zhou et al.	1

Respecto al nivel de adopción o compatibilidad de WASI 0.2 (junto con el Wasm Component Model), se puede diferenciar entre soporte en lenguajes de programación y en Wasm *runtimes*. Sobre el primer grupo, los compiladores de Wasm específicos de cada lenguaje deben soportar estas nuevas funcionalidades para generar un binario Wasm totalmente compatible sin necesidad de adaptaciones manuales como la inclusión de marcadores en el código, que era justamente lo que ocurría previamente en WASI 0.1. Por ejemplo, en el caso de Go, el compilador TinyGo [199] proporciona soporte nativo para WASI 0.2 a partir de su versión 0.34.0. Asimismo, también es interesante observar las herramientas adicionales que proporciona cada lenguaje para el desarrollo de componentes. En el caso de Go, la herramienta *wit-bindgen-go* genera *bindings* (una manera de conectar dos componentes diferentes de *software* para que puedan trabajar conjuntamente, aunque estén desarrolladas en diferentes lenguajes o entornos) en Go para las interfaces definidas en WIT y finalmente generar un componente. En resumen, después de realizar el debido análisis, se puede afirmar que Rust es el lenguaje con mayor soporte de WASI 0.2 y el Component Model, seguido por Go, JS y Python. Desafortunadamente, el nivel de implementación actual de WASI se encuentra en muchos casos lejos del supuesto ideal en el que código de fuente nativo, que utiliza librerías intrínsecas a los lenguajes, se puede compilar directamente a un componente WASI sin realizar acciones o adaptaciones manuales. Aquí es donde entran en juego los anteriormente descritos generadores de *bindings* WIT (*wit-bindgens* [200]), que ayudan a crear componentes completos, junto con librerías que ya implementan las interfaces WIT que especifican las funcionalidades de WASI 0.2, por ejemplo, para realizar llamadas HTTP o actuar como *proxy*.

En lo que respecta a los *runtimes* (que se describirán con más detalle en el siguiente subapartado), éstos han de implementar en su código fuente las APIs de WASI definidas en su versión 0.2 para ser compatibles con ella, de manera que puedan ejecutar los componentes Wasm resultantes de las compilaciones anteriores. Es por ello que en el siguiente apartado 2.7.2.2 se podrá ver con más detalle el alcance de esta compatibilidad.

En cuanto al futuro de la estandarización de WASI, la primera actualización de WASI 0.2 (o 0.2.1) supuso otro hito importante, ya que definió el procedimiento de actualización, así como el ritmo para realizar actualizaciones incrementales [201]. El subgrupo de trabajó acordó aumentar la frecuencia y acotar el ámbito de estas actualizaciones con el objetivo de facilitar la implementación de WASI en *runtimes* y en las herramientas ofrecidas por los lenguajes de programación. De esta manera, la intención es lanzar una versión de WASI cada dos meses, siempre asegurando retrocompatibilidad con las versiones anteriores. Además, esta primera actualización trajo consigo el anuncio de la publicación de una especificación para empaquetar componentes de WebAssembly en imágenes OCI [202] junto con una serie de herramientas para los desarrolladores [203], suponiendo un paso adelante en la coexistencia de Wasm con otros tipos de virtualización como los contenedores (esta temática se abordará con más detalle en el apartado 2.7.3). Ampliando el

horizonte hasta el futuro a medio plazo, la siguiente gran actualización de WASI será el lanzamiento de la versión 0.3.0 (WASI 0.3), prevista inicialmente para finales de marzo de 2025 pero que a principios de abril (momento en el que se ha cerrado la escritura de esta tesis) aún no ha sido lanzada, en la que se prevé incluir compatibilidad nativa con las comunicaciones asíncronas tanto a nivel de funcionamiento como de composición de módulos Wasm (se pretende añadir los tipos *future* y *stream* a WIT). A más largo plazo, se espera que todo este trabajo cristalice en la versión definitiva WASI 1.0, cuyo lanzamiento está previsto para 2026 [204], con un sistema completamente estandarizado. La versión más reciente de WASI en el momento de la finalización de la escritura de esta tesis es la 0.2.5, cuyo lanzamiento se produjo en abril de 2025.

Para concluir con la detallada descripción que se ha realizado sobre el desarrollo y estandarización de WASI, que como se ha visto no es un proceso para nada trivial, se puede apuntar que este avance en la estandarización de WASI acerca Wasm al sueño o predicción de Solomon Hykes sobre el esperanzador futuro de esta tecnología en el mundo de la computación, dado que se trata de la pieza necesaria para habilitar completamente esta tecnología en el lado del servidor como alternativa de virtualización.

Finalmente, aunque como se ha visto anteriormente WASI ya soporta el empaquetado de componentes en estándares OCI, el SIG (Special Interest Group) de empaquetado de la Bytecode Alliance está desarrollando en paralelo su propio protocolo de registro (o repositorio) para el almacenamiento y distribución de paquetes Wasm llamado *warg* [205]. Este protocolo está completamente orientado al Wasm Component Model y diseñado para habilitar la federación de múltiples registros independientes para facilitar la interoperabilidad en la ejecución de módulos Wasm en entornos distribuidos.

2.7.2.2. Wasm *runtimes*

Al igual que los *runtimes* de contenedores, y como queda plasmado en la arquitectura de bloques de la Figura 2.14, los *runtimes* de WebAssembly son un elemento esencial del *stack* de Wasm porque, en última instancia, la ejecución de los módulos Wasm depende completamente de ellos. Es por este motivo que en los últimos años se han desarrollado varios *runtimes* de Wasm que son capaces de ejecutar aplicaciones virtualizadas en Wasm fuera de los navegadores web. Por lo que concierne a esta tesis doctoral, el aspecto más relevante de estos novedosos *runtimes* es que sus desarrollades comparten una visión común sobre el potencial de Wasm en el ámbito del *edge computing*, por lo que han decidido adaptar estos *runtimes* para su ejecución en infraestructura que cuenta con las características típicas de esta capa del continuo. Estos *runtimes* difieren entre sí en el soporte de funcionalidades de WASI, aunque a partir del punto de inflexión que ha supuesto WASI 0.2 se podría hablar directamente del soporte de esta especificación (realmente WASI 0.2 actualmente solo está completamente soportado por

Wasmtime, que está desarrollado por la misma Bytecode Alliance), así como en la provisión de SDKs (*Software Development Kit*) de programación o APIs de implementación que permiten a los programadores cargar módulos Wasm en el código fuente de sus aplicaciones, entre otros aspectos. Otra característica interesante que se ha identificado después de analizar la documentación de varios de los *runtimes* más populares (Wasmer [206], Wasmtime [207] y WasmEdge [208]]), es que aportan compatibilidad nativa con los *runtimes* de contenedores de bajo nivel crun y youki (descritos en la sección 2.6.3) con el objetivo de lograr una integración con aplicaciones desplegadas usando contenedores, una tendencia que se explicará en detalle en el siguiente apartado.

Además, este creciente interés por el uso de WebAssembly (Wasm) se refleja en iniciativas como la de GraalVM [209], una distribución del JDK (Java Development Kit) de alta eficiencia desarrollada por Oracle, empresa que también es mantenedora del lenguaje Java. GraalVM ha logrado captar la atención de la comunidad de Java por permitir la compilación *ahead-of-time* (AOT), es decir, en el momento de la construcción del programa en lugar de en el momento de ejecución (JIT, *just-in-time*). En este contexto, GraalVM ha desarrollado su propio *runtime* de Wasm (GraalWasm [210]) para permitir embeber la ejecución de módulos de Wasm desde un programa escrito en Java.

Realizar una comparación descriptiva de los diferentes *runtimes* empleando únicamente texto resultaría en un contenido tedioso y de innecesaria profundidad para el alcance de esta tesis doctoral, puesto que su documentación propia está disponible de manera abierta en Internet, habilitando al lector de esta tesis a profundizar en sus características y funcionalidades únicas. Es por este motivo por el que se ha decidido ilustrar las diferencias entre los *runtimes* existentes mediante una tabla comparativa (Tabla 2.7), un material más visual y clarificador que además se realiza con la intención de que se convierta en un recurso útil para los desarrolladores a la hora de elegir un *runtime* en el que ejecutar los módulos Wasm desarrollados.

Finalmente, dado que esta memoria comprende un trabajo de investigación en sí mismo, es importante destacar algunos trabajos de investigación relacionados con los *runtimes* de Wasm, aunque se ha decidido no incluirlos en la Tabla 2.7. En [192] se presenta Wasmachine, un sistema operativo que intenta trasladar la especificación WASI a su *kernel* escrito en Rust (las ventajas de Rust en comparación con C ya se desarrollaron en el punto previo 2.6.3), con el objetivo de evitar la sobrecarga de las llamadas al sistema introducidas por el *runtime* de Wasm, para finalmente lograr un mayor rendimiento en comparación con el código nativo ejecutado directamente sobre un OS basado en Linux. Los resultados presentados en este artículo muestran un aumento en la velocidad de ejecución del 21%. Por otro lado, Twine es un *runtime* de Wasm como resultado de la interacción entre el mundo empresarial y el proyecto de investigación H2020 VEDLIoT, ya que se basa en WAMR, pero realmente está más enfocado a entornos embebidos y confiables [211].

Tabla 2.7: Comparación de los Wasm *runtimes* más populares

Tecnología	Lenguaje de desarrollo	WASI	Lenguajes en los que puede ser embebido	Arquitecturas de CPU soportadas	Uso en otros productos o en entornos de despliegue reales	Responsable	Popularidad
Wasmer	Rust	Extensión propia de 0.1: WASIX	C, C++, C#, D, Dart, Go, Java, JavaScript, PHP, Postgres, Python, Ruby, Rust, OCaml, Zig	x86_64, ARMv8, RISC-V	crun, Fluence Labs	Wasmer Inc.	19.1k *stars* 819 *forks*
Wasmtime	Rust	Totalmente compatible con 0.2	Bash, C, C++, Go, .NET, Python, Ruby, Rust	x86_64, ARMv8, RISC-V, S390x	crun, Shopify, Fastly, DFINITY, Microsoft, wasmCloud, Spin	Bytecode Alliance	15.5k *stars* 1.3k *forks*
WasmEdge	C++	0.1, trabajando en el soporte de 0.2	C, Go, Node.js, Python, Rust	x86_64, ARMv8, RISC-V	crun, Docker Engine, Suborbital scheduler, Suborbital engine	CNCF Sandbox	8.6k *stars* 771 *forks*
Wasm3	C	0.1	Arduino, C, C++, Go, JavaScript, Perl, Python, .NET, Nim, Swift, Zig	x86_64, x86_32, ARMv8, ARMv7, RISC-V, MIPS, ARC	Shareup, wasmCloud	CNCF Sandbox	7.3k *stars* 464 *forks*
WAMR	C	0.1	C, C++, Go, Python, Rust	x86_64, x86_32, ARMv8, ARMv7, RISC-V, MIPS, ARC	Envoy, Faasm, Inclavare Containers	Bytecode Alliance	5k *stars* 628 *forks*

wazero	Go	0.1	Go	x86_64, ARMv8, RISC-V64	Trivy, WeScale, YoMo	Tetrate Labs	5k *stars* 258 *forks*
Lunatic	Rust	Trabajando en su implementación	AssemblyScript, Rust	x86_64	No documentado	Lunatic solutions	4.6k *stars* 137 *forks*
WAVM	C/C++	0.1	C, C++	x86_64	No documentado	CNCF Sandbox	2.7k *stars* 225 *forks*
GraalWasm	Java	0.1	Java	x86_64, ARMv8	No documentado	Oracle	(repositorio de GraalVM) 20.7 *stars* 1.7 *forks*

2.7.2.3. Orquestación de módulos Wasm

Al igual que ocurre con los contenedores, se hace patente la necesidad de un sistema de orquestación para organizar el despliegue y la interacción de módulos Wasm, y de esta manera avanzar hacia plataformas tecnológicas inteligentes y completas. La existencia de este tipo de plataformas de despliegue avanzadas se puede ver como otro indicador de la importancia de WebAssembly y la cantidad de esfuerzos que se están poniendo sobre este, sobre todo en comparación de las MicroVMs (aunque sí que existen orquestadores de VMs como OpenStack) o Unikernels, los cuales carecen de plataformas de orquestación específicas.

Spin es una plataforma de código abierto desarrollada por Fermyon cuya finalidad es la ejecución de componentes Wasm (utilizando Wasmtime como *runtime*) como respuesta a ciertos eventos, siendo equivalente a una plataforma *serverless* para el despliegue de este tipo de cargas de trabajo [212]. Las aplicaciones en Spin son el resultado de la unión de varios componentes (realmente se tratan de módulos Wasm) que se especifica en un archivo TOML (*spin.toml*) llamado *manifest*, en donde además se describen los eventos (*triggers*) a los que debe reaccionar esta aplicación, e incluso los permisos de red de estos componentes, ya que por defecto no tienen permitido realizar peticiones de red al exterior debido a que Spin intenta ser coherente con el diseño general de Wasm en el que los permisos para utilizar ciertas funcionalidades se deben otorgar explícitamente. Asimismo, Spin proporciona un SDK para varios lenguajes que soportan WASM como Python, JavaScript, Go o Rust, que ofrece una serie de *templates* para la creación de aplicaciones o de componentes para ser posteriormente añadidos a las aplicaciones, previo paso por el proceso de compilación a Wasm del lenguaje elegido. Este proceso utiliza mayoritariamente módulos de WASI 0.1, sin embargo, Spin también soporta el WASM Component Model de manera nativa, dado que permite la inclusión de componentes Wasm como dependencias de sus aplicaciones, siempre que estos componentes no tengan a su vez más dependencias de otros componentes y que estén publicados en un registro externo. De esta manera, Spin es capaz de descargarse el componente Wasm de su registro, localizar el *export* definido en la dependencia, y enlazarlo en los *imports* del componente Wasm matriz de la aplicación de Spin, hablando en términos de WIT y del Component Model. En cuanto a la mejora de rendimiento ofrecida por Wasm en comparación con los contenedores para la ejecución de funciones, en el trabajo [213] se puede observar como mejora el tiempo de instanciación de funciones al utilizarse Wasm.

Entre los sistemas de orquestación diseñados específicamente para controlar el despliegue de módulos Wasm claramente destaca *wasmCloud*, que además de proporcionar una tecnología de orquestación de módulos Wasm, también aporta herramientas a los desarrolladores para construir y componer (incluso en tiempo de ejecución gracias a su soporte del Component Model) componentes Wasm, así como distribuirlos para compartirlos con la comunidad y de esta manera fomentar su reutilización [214]. wasmCloud está basado en el Component Model (descrito en el

anterior apartado 2.7.2.1) de manera que permite extender aplicaciones con funcionalidades proporcionadas por el estándar WASI (http, *random*...) o por wasmCloud directamente (SQL, mensajería...). Asimismo, esta herramienta permite ejecutar tanto módulos Wasm empaquetados siguiendo el estándar OCI (como los contendores) o utilizando módulos de WebAssembly locales (archivos *.wasm*). En cuanto a la orquestación de módulos Wasm, el motor encargado de ésta (*wasdm*) se basa en técnicas de control del estado de los módulos mediante la ejecución de una función de reconciliación, al igual que los Operadores de K8s, no siendo la única similitud que comparte con K8s, ya que los módulos Wasm se definen en manifiestos YAML siguiendo el formato Open Application Model (OAM), que se trata de un formato definido por el proyecto de la CNCF KubeVela orientado a la definición de aplicaciones en lugar de contenedores como en el caso de los manifiestos de K8s [215]. No obstante, la sinergia entre ambas tecnologías de orquestación es tal que (incluso podría verse como un adelanto del contenido de la siguiente subsección) wasmCloud puede instalarse en K8s mediante el despliegue de un Operador, por lo que se podría gestionar el despliegue de los módulos Wasm en un clúster mediante el uso de *Custom Resources* (CRs) de K8s. Este Operador ejecuta dos funciones de reconciliación: en primer lugar, la propia del Operador de K8s, que transmitiría la orden de ejecución al motor *wasdm,* y seguidamente la propia de este componente para desplegar finalmente el componente Wasm en el clúster de K8s.

wasmCloud crea su propia infraestructura mediante la definición e instalación de nodos, en los que se instala el CLI *wash* (wasmCloud Shell) junto con el runtime Wasmtime. De igual manera, wasmCloud define su propia malla de red o de servicios llamada *Lattice* para permitir la comunicación entre componentes de Wasm, que está basada en la creación de clústeres federados del *broker* NATS para ser compatible con la heterogeneidad de redes del continuo *Cloud-Edge-IoT* [216].

2.7.3. Coexistencia de diferentes tipos de virtualización

Como se ha repetido varias veces durante este estado del arte, la existencia de varias técnicas de virtualización no significa que unas técnicas deban ser reemplazadas por otras, o que solo un tipo de virtualización es válida, sino que aplicaciones virtualizadas con diferentes técnicas pueden incluso coexistir en la misma máquina o en el mismo entorno de ejecución. Realmente, la elección de la tecnología de virtualización debe realizarse en función del caso de uso final de la aplicación. Por ejemplo, los contenedores son una mejor opción para aplicaciones que necesitan un control de sistema de archivos robusto o ejecutar operaciones de E/S intensivas, mientras que Wasm sería una mejor elección en el caso de servidores web simples [217]. Además, como un primer paso hacia esta simultaneidad de tipos de cargas de trabajo, Docker (recordemos, el estándar *de facto* para ejecutar contenedores) anunció a finales de 2022 la compatibilidad del Docker Engine con

Wasm mediante el uso de WasmEdge como su Wasm *runtime* para ejecutar este tipo de cargas de trabajo [218]. El requisito para poder llevar este tipo de despliegues a cabo es que los desarrolladores deben asegurarse de que los módulos Wasm estén empaquetados como imágenes compatibles con el estándar OCI , de modo que el *runtime* de alto nivel del Docker Engine (containerd) pueda manejar adecuadamente el módulo Wasm a través de un *shim* (en el ámbito de los contenedores, un *shim* es un proceso ligero que actúa como intermediario entre un *runtime* de contenedores y el mismo proceso del contenedor, cuya finalidad es facilitar el control del ciclo de vida del contenedor) llamado *containerd-wasm-shim*, que fue creado específicamente con este propósito. En esta primera versión, la compatibilidad nativa era completa, ya que este *shim* interactuaba en última instancia con el *runtime* de WasmEdge para ejecutar una aplicación Wasm de la misma manera que interactuaría con el *runtime* de contenedores de bajo nivel runC para ejecutar un contenedor, tal y como se ilustra en la anterior Figura 2.6.

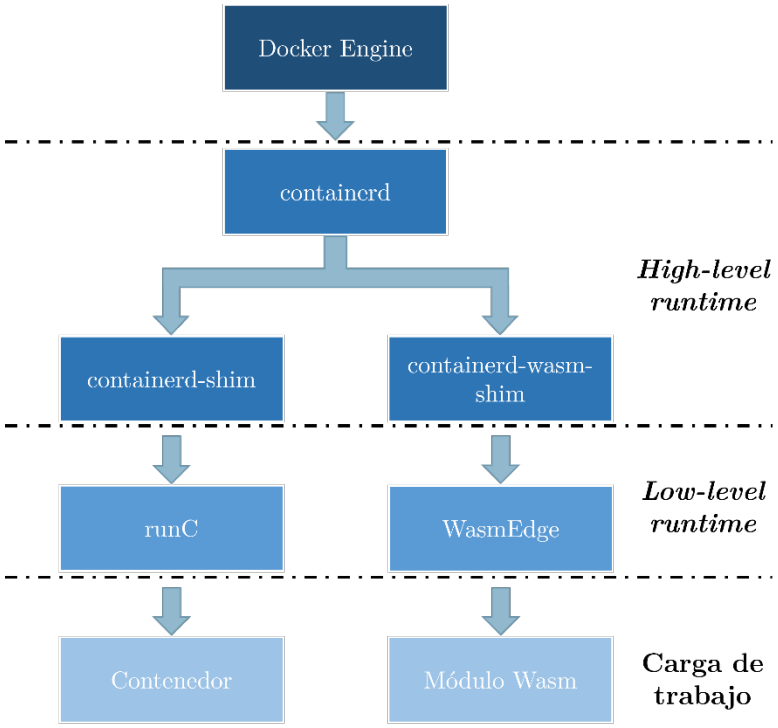


Figura 2.16: Arquitectura del Docker Engine para ejecutar módulos Wasm

A partir de este anuncio y después de varios meses de testeo por parte de los usuarios, en marzo de 2023 Docker lanzó una actualización de su integración con Wasm en la que extendió la lista de *runtimes* compatibles con el soporte de Spin, slight y Wasmtime [219]. Además, fruto de esta extensión reveló el proyecto de código abierto *runwasi*, que se trata de una librería escrita en Rust para facilitar el desarrollo de *shims* de containerd (la creación de un *shim* específico para cada *runtime* es necesaria) para ampliar su compatibilidad, y por ende la de Docker, con un mayor número de *runtimes* de Wasm [220]. Asimismo, esta librería hace posible la adaptación de cualquier *runtime* de Wasm con containerd, ya que proporciona

una estructura base para que los desarrolladores se centren en implementar las funcionalidades requeridas para cada acción. Actualmente, los *runtimes* soportados oficialmente por containerd son: Wasm, Wamr, Wasmedge, Wasmer y Wasmtime, de los cuales se ha publicado el código de sus *shims* en el repositorio anteriormente introducido de *runwasi*.

De una manera similar, el *runtime* de contenedores de alto nivel CRI-O también aporta compatibilidad con el despliegue de módulos Wasm, pero siguiendo una estrategia diferente. CRI-O delega la ejecución de los módulos Wasm en su *runtime* de bajo nivel por defecto crun, que a su vez ofrece compatibilidad de manera intrínseca con diversos Wasm *runtimes* tal y como se ha descrito en secciones anteriores. Esta estrategia también podría ser utilizada en containerd, aunque se avista como mejor opción la solución basada en *shims* recomendada por su documentación oficial.

El empaquetado de módulos Wasm en imágenes OCI, cuyo proceso fue estandarizado y ampliamente mejorado a partir de WASI 0.2.1, permite abrir un gran abanico de posibilidades para la coexistencia de diferentes cargas de trabajo que realmente se han virtualizado utilizando diferentes técnicas de virtualización (no solo Wasm o contenedores), ya que permite abstraer el despliegue o ejecución de estas cargas de trabajo, siempre que la infraestructura esté correctamente configurada para soportar la ejecución de dichos servicios. Este aspecto es de suma importancia en entornos heterogéneos como el continuo de computación sobre el que se fundamenta esta tesis. No obstante, este escenario de coexistencia plantea un desafío de mayor dificultad: el requisito de un mecanismo de orquestación común. En la sección 2.6.5.2 se han estudiado múltiples tecnologías para demostrar que K8s está siendo cada vez más adaptado y utilizado exitosamente como tecnología de orquestación de contenedores en el *edge*, por esta razón el siguiente paso natural sería analizar si K8s encaja para este propósito y pudiera utilizarse de igual manera para orquestar otros tipos de cargas de trabajo.

Observando en detalle la arquitectura general de Kubernetes, el componente *kubelet* (recordemos que su misión es asegurarse de que los contenedores se ejecutan dentro de *Pods*) interactúa con un *runtime* de contenedores de alto nivel (también llamado *runtime* CRI o implementación CRI en K8s) que implementa la *Container Runtime Interface* de Kubernetes (CRI), actuando como cliente de esta API gRPC para en última instancia controlar el despliegue de Pods y sus contenedores [104]. Profundizando en el funcionamiento de K8s, se puede afirmar que el recurso llamado *RuntimeClass* (se define como "una función para seleccionar la configuración del *runtime* de contenedores" [221]) es el componente subyacente clave para controlar el comportamiento del *kubelet*. Por ende, este recurso puede utilizarse debidamente con el objetivo de mapear diferentes tipos de cargas de trabajo con su *runtime* de bajo nivel correspondiente, situándose en el mismo nivel que el *runtime* de bajo nivel de contenedores de la Figura 2.6, si es visto desde un punto comparativo a nivel de bloques de arquitectura. Como ejemplo práctico, mediante la creación y configuración de las debidas *RuntimeClasses*, CRI-O o

containerd (como *runtime* de alto nivel), serán capaces de enviar una solicitud al *runtime* de bajo nivel adecuado para ejecutar la carga de trabajo solicitada, como se muestra en la Figura 2.17 para el caso de contenedores, Kata Containers, unikernels de Unikraft y módulos de WebAssembly. El único requisito para la puesta en funcionamiento de este escenario de orquestación es el empaquetado previo de la aplicación en una imagen compatible OCI, que tiene que estar previamente almacenada en un registro de imágenes OCI, ya que de esta manera el *runtime* de alto nivel será capaz de descomprimir la imagen para seguir con el proceso de ejecución. A nivel práctico, la implementación de esta idea se puede observar en el Operador de K8s *runtime-class-manager*, cuya finalidad es facilitar la gestión de las anteriormente descritas *RuntimeClasses* e incluso la instalación del Wasm *runtime* correspondiente en los nodos del clúster [222].

Además, como se ha avanzado anteriormente, tanto WasmCloud como Spin pueden instalarse directamente sobre K8s, por lo que también permiten el despliegue de módulos Wasm directamente sobre K8s mediante el despliegue de un CR propio de WasmCloud (tipo *Application)* o de Spin (tipo *SpinApp*), que es interpretado por el controlador respectivo para ocuparse de la ejecución del módulo descrito por parte del *runtime* Wasmtime en última instancia.

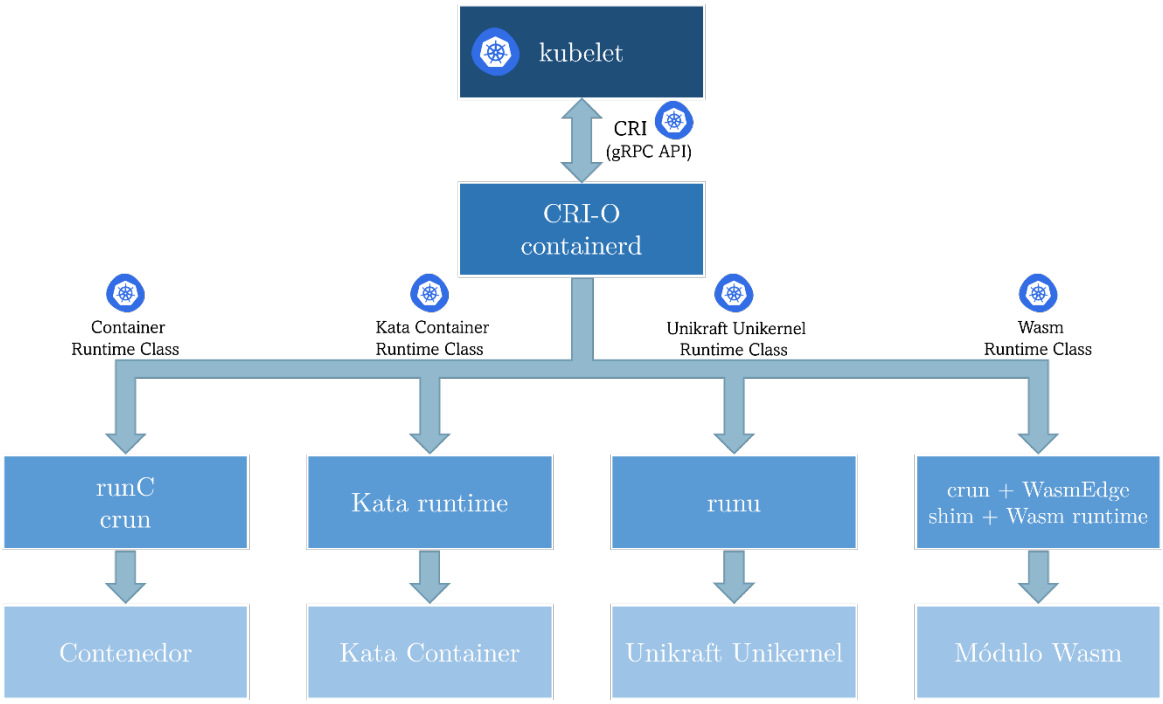


Figura 2.17: Ejecución de diferentes cargas de trabajo en K8s

De igual manera, es interesante realizar una descripción de algunos proyectos y tecnologías que encajan perfectamente dentro del contexto del despliegue de módulos Wasm en clústeres de K8s. Krustlet es un proyecto experimental que tiene como objetivo la reescritura del componente *kubelet* de K8s utilizando el lenguaje

Rust para conseguir la ejecución nativa de cargas de trabajo basadas en Wasm. Además, Krustlet utiliza las tolerancias de K8s (un conjunto de requisitos que han de cumplir los nodos para desplegar *Pods* en ellos) para orquestar y asignar dichas cargas de trabajo a nodos de K8s, teniendo como requisito la previa instalación del *runtime* Wasmtime en dichos nodos [223]. Como apunte final, la relación entre K8s y los módulos Wasm no se limita al despliegue de aplicaciones Wasm en lugar de contenedores. Algunos trabajos de investigación innovadores, como el presentado en [224], proponen reducir la sobrecarga del plano de control de K8s, una de las principales razones de su gran tamaño, mediante el despliegue de Operadores de K8s (recordemos, se tratan de unos módulos que extienden las funcionalidades de la API de K8s) basados en Wasm bajo demanda. Este trabajo mostró resultados reveladores a la vez que prometedores, ya que el reemplazo de controladores tradicionales basados en contenedores por controladores basados en Wasm resultó en una reducción del consumo de memoria de alrededor del 64%.

Para concluir esta sección, es necesario poner en conocimiento que la orquestación de servicios no es el único desafío al que se enfrenta el intento de diseñar un sistema que pretenda controlar un ecosistema heterogéneo, tanto en cuanto a elementos de computación como a diferentes tipos de cargas de trabajo, como es el continuo *Cloud-Edge-IoT*. Otras problemáticas que se deben tener en consideración son la abstracción de la capa de red o del intercambio de datos, aún más en sistemas descentralizados o federados como el descrito. Estos desafíos se han abordado en numerosos proyectos o artículos de investigación, como el proyecto aerOS (que se describirá en el siguiente capítulo) o el artículo de congreso [225], en los que ha participado el autor de esta tesis doctoral. En dicho artículo, se realizó una propuesta teórica de una arquitectura para desarrollar un sistema de control de cargas de trabajo heterogéneas (contenedores, módulos Wasm) en el continuo de computación, basado en técnicas de AI y en la federación de los elementos de computación mediante el uso de un *context broker* (ver apartado 2.4.1). Además, el doctorando también ha participado en un artículo en el que se argumenta a favor del potencial de Wasm para la solución de algunos de los problemas actuales en el ámbito de la ingeniería del *software*, como el desarrollo de aplicaciones multi arquitectura o la dificultad para reducir su tiempo de despliegue en el continuo de computación [226].

Capítulo 3

Transición hacia el continuo *Cloud-Edge-IoT*

3.1.Introducción

En el capítulo anterior se ha realizado un extensivo estudio sobre el marco teórico y las herramientas y productos tecnológicos creados a partir del mismo, sobre los que se sustenta esta tesis doctoral. No obstante, esto puede ser visto como una descripción aislada de tecnologías y herramientas sin que se llegue a visualizar de manera evidente las potenciales relaciones entre ellas, de manera que puedan formar sistemas robustos para la consecución de un objetivo general. Por lo tanto, aparece la necesidad de plasmar en un capítulo independiente las construcciones realizadas que se apoyan en estas tecnologías, englobándolas en un marco general basado en los paradigmas de computación dominantes y en su constante evolución hasta finalizar en el actual paradigma del continuo *Cloud-Edge-IoT*, ya que es el pilar fundamental de todo este trabajo.

Esta tesis doctoral se nutre significativamente de años de experiencia en el mundo de la investigación científica abierta y colaborativa entre varios equipos, específicamente en proyectos colaborativos de investigación a nivel estatal y europeo, por lo que se ha decidido dedicar esta sección a recapitular dichos proyectos de investigación y su influencia en esta tesis doctoral, a pesar de que se también se hayan descrito varias soluciones propietarias o de mercado, como por ejemplo, la oferta de productos para el *edge computing* de los grandes dominadores del mercado de la computación en la nube.

La estructura de cada sección parte de la introducción del marco o paradigma tecnológico que se pretende abordar, seguida por la descripción de diferentes enfoques de implementación en cada proyecto de investigación. Entrando en más detalle, se realiza una descripción de un proyecto de investigación principal en el que se plasma la idea principal, siendo también en el que el doctorado ha tenido más participación, debido a que puede aportar un mayor conocimiento global a esta tesis que otros proyectos similares en los que solo se ha podido acceder a su documentación, de una manera externa. A este proyecto principal se le añaden unas propuestas de mejora con el estado actual de la tecnología descrita en el estado del arte junto con los conocimientos clave adquiridos por el doctorando después de varios años de investigación, siempre teniendo en cuenta el contexto en que fueron desarrollados.

Finalmente, todo este capítulo se resume en una tabla comparativa de los diferentes proyectos de investigación analizados para proporcionar al lector un material más visual en el que apoyarse, pudiendo usar también esta tabla como punto de entrada a los aspectos más interesantes de cada proyecto.

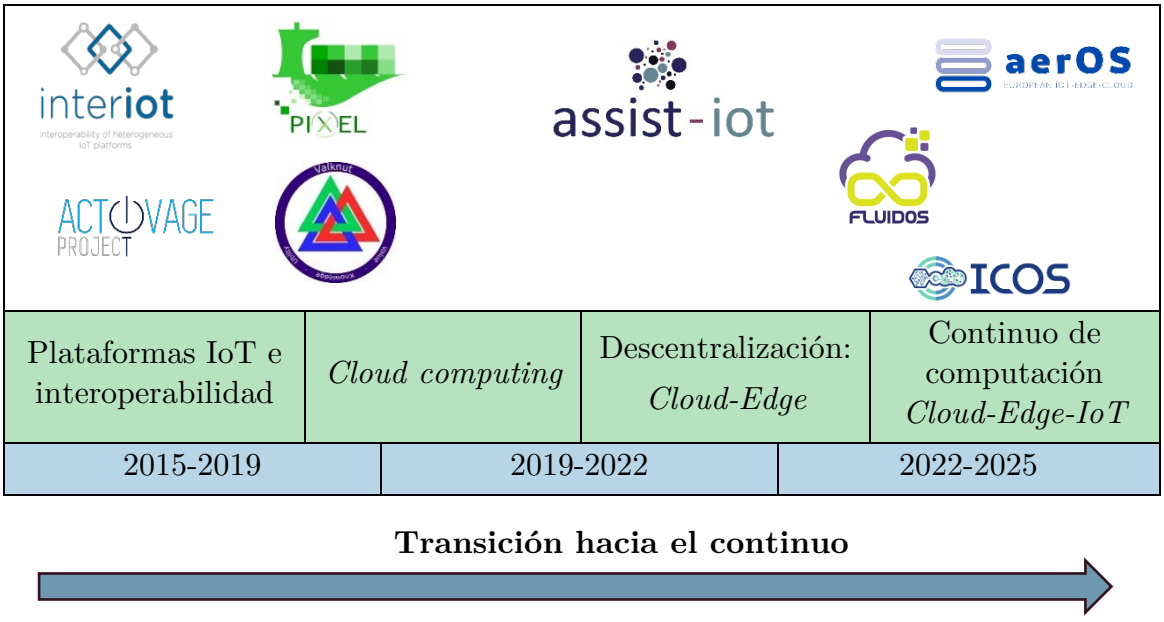


Figura 3.1: Transición hacia el paradigma de computación

3.2.Plataformas IoT centralizadas

El periplo del doctorando en el mundo de la investigación empezó en 2016 gracias a la concesión de una beca de colaboración con el grupo de investigación SATRD. El trabajo durante estas becas se centraba en el desarrollo de elementos de la arquitectura de diferentes proyectos de investigación, así como del proceso de *testing* de diferentes elementos de los que no se tenía conocimientos previos para profundizar en su usabilidad y continua mejora. Generalmente, todo este trabajo se englobaba en entornos de plataformas IoT.

A partir de mediados de la década de 2010, el *Internet of Things* era uno de los principales campos de investigación e innovación en el mundo IT, ya que la mejora de las redes de comunicaciones y de la electrónica de sistemas embebidos se tradujo en el aumento del número de dispositivos conectados a internet. En este ámbito, empezaron a desarrollarse las llamadas plataformas IoT, cuyo objetivo era la definición y creación de un marco tecnológico que facilitara la configuración y gestión de estos nuevos elementos que se estaban conectando a la red. Estas plataformas permitían recopilar, procesar y transmitir datos desde estos dispositivos para automatizar procesos, monitorear actividades y generar información útil en tiempo real, contando muchas de ellas con gestores de contexto como punto de partida para gestionar la información contextual. Claros ejemplos de estas plataformas son la anteriormente descrita FIWARE, así como universAAL IoT [227] o Sofia2 [228], entre muchas otras. Sin embargo, la proliferación de diversas plataformas sin una estandarización previa provocó el surgimiento de un nuevo problema de interoperabilidad, puesto que cada plataforma definió su propia estructura de APIs y de modelos de información [229]. Ante esta problemática, la Comisión Europea intentó reaccionar impulsando una convocatoria de proyectos de investigación orientados mejorar la interoperabilidad entre distintas plataformas IoT. El objetivo último era permitir que los usuarios finales seleccionaran la plataforma que mejor se adaptara a sus necesidades sin renunciar a su integración en soluciones más amplias y globales, puesto que ya no era posible la especificación de un estándar común para este tipo de plataformas. Ejemplos destacados de esta iniciativa son los proyectos H2020 Inter-IoT [230]o H2020 ACTIVAGE [231], en los que participó el doctorando durante sus primeros compases en el mundo de la investigación.

Al mismo tiempo que se desarrollaban las citadas plataformas IoT, comenzaba a tomar fuerza la tendencia a centralizar los recursos de computación y el despliegue de servicios en centros de datos especializados gestionados por empresas líderes del sector como Google, Microsoft o Amazon, paradigma que se acabó conociendo como computación en la nube, y que conllevó el desarrollo de grandes avances como la virtualización utilizando contenedores, la cual acabaría convirtiéndose en prácticamente un estándar (véanse las secciones 2.2 y 2.3). Este hecho hizo que las iniciativas de investigación también centraran parte de sus esfuerzos en adoptar y mejorar estas tecnologías *cloud-native* para mejorar los sistemas basados en el IoT,

surgiendo una especie de sinergia en la que mientras que la fuente de datos seguía asociada a la capa de arquitectura del IoT, la posterior computación asociada a estos datos (análisis de datos, reacción a eventos, etc.) fuera trasladada a la nube con el objetivo de explotar sus tecnologías en auge [32]. Paralelamente, las empresas anteriormente mencionadas empezaban a lanzar sus plataformas propias de computación en la nube con el objetivo de abstraer a los usuarios finales de la complejidad tecnológica subyacente, en las que naturalmente incluyeron herramientas enfocadas al IoT. Sin embargo, en la mayoría de estas iniciativas de investigación el objetivo no se limitaba a usar una plataforma o herramienta ya desarrollada y dependiente del proveedor de *cloud*. En su lugar, se apostó por desplegar infraestructura propia en los diferentes casos de uso (una estrategia de despliegue conocida como *cloud on-premise*) con el propósito de investigar las tecnologías *cloud-native* con licencia *open source* que se preveía que iban a mejorar los resultados de dichos casos de uso, y que al mismo tiempo también estaban siendo adoptadas por las mismas empresas líderes en el sector.

En este último contexto encajan el proyecto europeo H2020 PIXEL y el proyecto estatal Retos-Colaboración VALKNUT, en los que el doctorando adquirió un rol más protagonista tras empezar a formar parte del personal de investigación contratado del grupo SATRD, poco después de finalizar su Máster en Ingeniería de Telecomunicación.

3.2.1. PIXEL

PIXEL (*Port IoT for Environmental Leverage*) fue una iniciativa financiada por la Comisión Europea a través del programa Horizonte 2020 (H2020), que se llevó a cabo entre mayo de 2018 y septiembre de 2021 [232]. Su principal propósito fue diseñar la primera plataforma flexible que combinara metodologías avanzadas junto con tecnología inteligente e innovadora para reducir el impacto medioambiental a la vez que optimizar las operaciones en puertos marítimos de tamaño mediano y pequeño, empleando el *Internet of Things* y técnicas nativas de la computación en la nube. La propuesta incluyó una arquitectura IoT completa, complementada con una serie de servicios que fueron probados en cuatro puertos europeos: Burdeos, Monfalcone, Tesalónica y El Pireo. Estos servicios se diseñaron para responder a las necesidades y desafíos del sector marítimo en el momento de su ejecución, como la previsión de la demanda energética, la anticipación de la congestión del tráfico, la integración de la logística intermodal y, de manera destacada, la mitigación del impacto ambiental. Desafortunadamente, muchos de estos objetivos siguen vigentes, puesto que el impacto ambiental de los centros logísticos, tanto marítimos como terrestres, siguen siendo uno de los desafíos que intenta solucionar la Agenda 2030 [233].

La arquitectura a alto nivel de PIXEL se divide en 3 bloques fundamentales más el bloque transversal de seguridad que afecta a cada uno de ellos [234]:

- Bloque de **adquisición de datos** o *Data Acquisition Layer* (DAL): este bloque se encarga de habilitar la entrada de información a la plataforma PIXEL. Los datos IoT de los sensores y los datos reales sobre las operaciones efectuadas sobre los barcos en el puerto son recogidos por elementos llamados agentes, que posteriormente los convierten a formato NGSI utilizando un modelo de datos adecuado para ser finalmente enviados al gestor de contexto Orion. Este bloque también gestiona la ejecución de estos agentes a partir de la configuración que han establecido los administradores de la plataforma.
- Bloque de **gestión de la información** o *Information Hub* (IH): la funcionalidad proporcionada por este elemento de la arquitectura es la persistencia de los datos de entrada a la plataforma en una base de datos de tipo NoSQL Elasticsearch [235], por lo que son proporcionados por el Information Hub. Además, proporciona funcionalidades extra para facilitar la obtención de estos datos por parte de los diferentes modelos y algoritmos ejecutados por las Operational Tools, así como para el almacenamiento de sus resultados de ejecución.
- Bloque de **herramientas operacionales** u *Operational Tools* (OT): se trata del elemento central de la arquitectura de PIXEL, puesto que su principal cometido es la orquestación de la ejecución de los diferentes modelos y algoritmos predictivos que han sido desarrollados para que sus resultados guíen a los operadores portuarios en el intento de solucionar o mejorar las problemáticas descritas en la descripción del proyecto. Estos modelos se ejecutan en forma de contenedores, y utilizan el módulo Information Hub tanto como su fuente de datos de entrada como de salida.

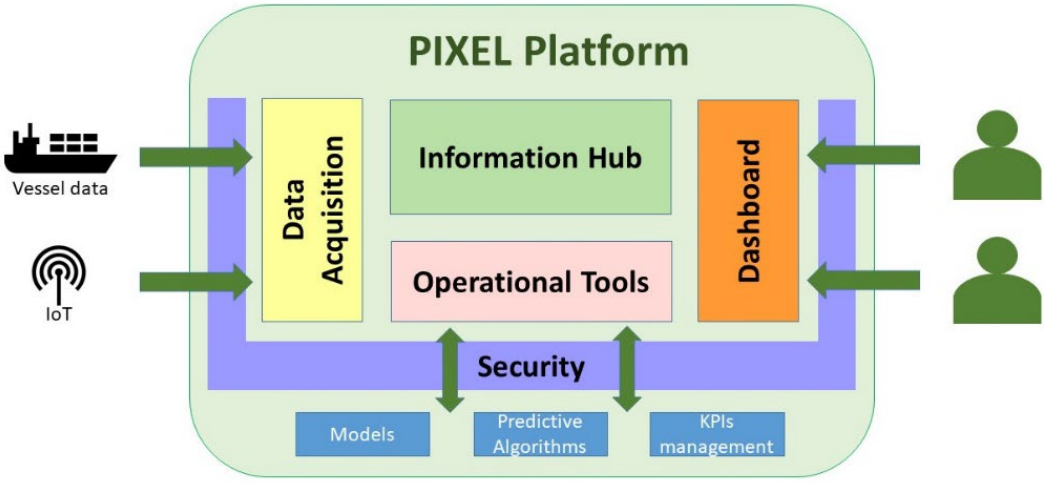


Figura 3.2: Arquitectura del proyecto PIXEL [234]

Además, también se incluye un *dashboard* o panel de control de manera que actúe como el punto de entrada único para el usuario final de la plataforma PIXEL,

desde el que se pueden configurar y ejecutar diversos agentes para nutrir a la plataforma con la información IoT y portuaria (AIS, *PortCalls*...) necesaria para permitir la ejecución de los modelos y algoritmos predictivos necesarios. Estos modelos también pueden ser lanzados manualmente o de manera más automática o controlada, dado que es posible planificar su ejecución en el tiempo. Asimismo, esta aplicación web permite consultar información relevante como los resultados de estos algoritmos o una serie de KPIs claves que proporciona la plataforma.

A nivel funcional, en la plataforma PIXEL todo el proceso empieza en el nivel de recogida de datos de las diferentes fuentes IoT y de los elementos de información del puerto. Uno de los primeros problemas a los que se enfrentó el proyecto fue la gran variabilidad en los métodos de recopilación de esta información en cada puerto, por lo que la creación y definición de un proceso común para la obtención y estandarización de la información de entrada era primordial. Esto se debe a que, por ejemplo, en algunos puertos la información sobre los barcos que actualmente están realizando operaciones en el puerto se podía obtener llamando a una serie de APIs REST, mientras que en otros se tenía que obtener de sistemas preexistentes heredados, como un listado en una página web. De manera similar, los puertos más modernizados disponían de una serie de sensores IoT para medir las emisiones de gases contaminantes cuyos datos era accesibles vía *brokers* de mensajería como MQTT, APIs o incluso se podían conectar al gestor de contexto Orion, pero en algunas situaciones incluso se debían de realizar estimaciones a partir de unos pocos datos. En consecuencia, en el proyecto se decidió crear una serie de agentes específicos para cada caso de uso, de manera que fueran capaces de obtener estos datos de su fuente, transformarlas a formato NGSI para en última instancia enviarlas a Orion para que fueran almacenadas en el Information Hub (un *wrapper* personalizado de una base de datos NoSQL con ciertas funcionalidades complementarias). Estos agentes se podían desarrollar en cualquier lenguaje de programación siempre y cuando cumplieran con las funcionalidades descritas, aunque se puso a disposición una librería común en Python llamada *pyngsi* [236] para facilitar la creación de entidades NGSI en Orion. Esto fue posible debido a que los agentes debían de empaquetarse como imágenes de contenedores usando unas etiquetas de configuración específicas (definidas como *LABELS* en el archivo Dockerfile). De igual manera, el módulo DAL proporciona una capa de orquestación y automatización adicional para configurar la ejecución de estos agentes haciendo uso de esta configuración y de su abstracción en contenedores [237].

El desarrollo de una plataforma IoT completa junto con una metodología común para la recopilación de datos de fuentes heterogéneas supuso un gran avance en el sector. Sin embargo, el corazón del proyecto PIXEL realmente reside en el desarrollo y ejecución de una serie de modelos y algoritmos predictivos que se alimentan de los datos almacenados en el módulo IH. Algunos ejemplos son: los modelos predictivos para estimar el tráfico de vehículos de entrada al puerto o la producción de energía fotovoltaica. Al igual que los agentes del módulo DAL, estos modelos y algoritmos pueden ser ejecutados con independencia del lenguaje de

programación en el que han sido desarrollados, puesto que el requisito necesario para formar parte de la plataforma es su empaquetado en imágenes de contenedores utilizando unas etiquetas especiales de configuración en las que se describen tanto sus datos de entrada como de salida utilizando un formato específico basado en JSON. De esta manera, se establece un modelo de configuración común que a su vez permite la configuración de su ejecución por parte de los usuarios en el *dashboard*, y es interpretable por el módulo OT, que se encarga de la ejecución de ellos vía el despliegue de contenedores. Este módulo OT internamente está compuesto por una API REST que actúa como puerta de entrada de las órdenes de los usuarios, y por un controlador interno que se encarga de traducir estas peticiones en acciones directas sobre los contenedores (ejecución, eliminación, descarga de imágenes, etc.) interactuando con Docker.

El proyecto PIXEL no solo fue relevante en cuanto al despliegue de una plataforma IoT completa totalmente enfocada al sector portuario utilizando patrones de diseño y tecnologías *cloud-native*, sino que gracias a que desarrolló uno de los modelos más ambiciosos para intentar medir la eficiencia medioambiental de los entornos portuarios marítimos: el *Port Environmental Index* (PEI) [238], en el que el doctorando participó de manera activa en su desarrollo e implementación tecnológica. Este modelo se basa en el cálculo retroactivo de un índice compuesto cuyo valor oscila entre 0 y 1, donde valores más cercanos a 1 indican una mayor eficiencia medioambiental del puerto. El PEI se calcula a partir de diferentes índices compuestos agregados, organizados mediante un árbol jerárquico con pesos definidos por el usuario. En el nivel inferior de este árbol se encuentran datos reales que incluyen desde mediciones de emisiones de gases contaminantes, como CO_2 y NOx, en distintas áreas del puerto (terminales, autoridad portuaria, barcos, etc.), hasta información detallada (modelo del motor, tipo de barco...) sobre los buques que realizan operaciones en el puerto. Para estructurar esta información, se definió un modelo de datos denominado eKPI (*Environmental Key Performance Indicator*), que se basa en el modelo de datos KPI de la iniciativa Smart Data Models de FIWARE [239]. En consecuencia, los agentes tienen que transformar los datos en bruto recibidos para ajustarse a este formato, garantizando así la coherencia y compatibilidad en el procesamiento de la información. Cabe destacar que la creación de este modelo necesitó un vasto trabajo previo de investigación por parte de expertos medioambientales, así como del cálculo eficiente de índices compuestos, que fue la temática principal de la tesis doctoral del codirector de esta misma tesis, Ignacio Lacalle [240]. Asimismo, el proceso de diseño, desarrollo e implementación final del PEI en puertos reales se plasmó en un artículo de investigación del que el doctorando figura como autor [238].

Propuesta de mejoras hacia el paradigma del continuo Cloud-Edge-IoT

Aunque el proyecto PIXEL apostó de un primer momento por utilizar tecnologías *cloud-native* e innovadoras en el momento de su realización, como lo son el uso de contenedores como herramienta de despliegue básica y común a todos

los componentes o el uso de bases de datos NoSQL, visto desde la perspectiva actual no cumpliría plenamente con dichos estándares de diseño. Esta circunstancia abre la puerta a plantear una serie de mejoras fácilmente implementables, que no solo evidencian los avances tecnológicos alcanzados en los últimos años, sino que también sirven como testimonio del rápido ritmo de innovación en este campo, un aspecto central que esta tesis doctoral busca abordar.

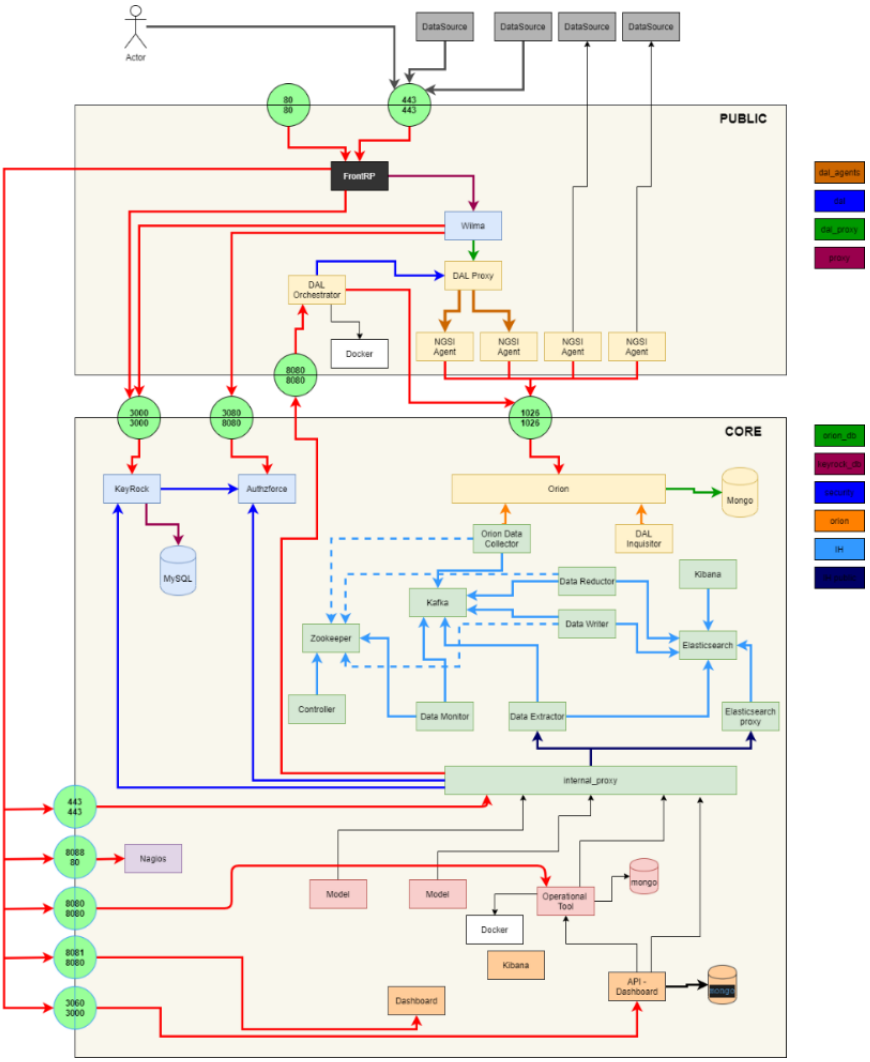


Figura 3.3: Componentes de la plataforma IoT de PIXEL [237]

Como se puede observar en la Figura 3.3, la instalación de PIXEL se realiza en dos máquinas diferentes. Esta separación responde a criterios de funcionalidad, permitiendo distribuir los diferentes módulos de la plataforma, que finalmente son desplegados utilizando contenedores. Entrando en detalles más técnicos, esta instalación se realiza mediante la ejecución de *scripts* personalizados que realmente ejecutan diversos ficheros de Docker Compose para desplegar los diferentes elementos (virtualizados en forma de contenedores) que componen la plataforma. Actualmente la elección más natural sería el despliegue de la plataforma en un clúster de Kubernetes previamente instalado, en el que no tendría mucho sentido realizar esta separación por máquinas, puesto que, aunque muy probablemente se

seguirían necesitando dos máquinas por motivos de recursos, en K8s se podría realizar esta misma separación de manera conceptual o virtual utilizando *namespaces*, o también optar por realizar la separación por bloques de la arquitectura. Asimismo, se podría reemplazar el uso de *proxies* o elementos de seguridad internos por políticas de red (*network policies*) en el mismo clúster, que pueden ser gestionadas por CNIs inteligentes como Cilium. Esta abstracción de los recursos de computación es claramente uno de los puntos fuertes del modelo de computación en la nube. Además, se podrían aprovechar los diferentes recursos que ofrece K8s para automatizar el despliegue, como el empaquetado de elementos usando Helm charts configurables, o la exposición de servicios junto con la obtención dinámica de certificados mediante el uso de *Ingress* y *cert-manager*.

Respecto al nivel operativo de la plataforma, la mayor modificación giraría entorno a la ejecución de los agentes del DAL y de los modelos y algoritmos predictivos de las OT. El concepto de ejecución de contenedores de PIXEL encaja completamente con el modelo conocido como *serverless* debido a que se resta importancia a la máquina que realmente ejecuta el contenedor que contiene la lógica necesaria. Como se ha descrito anteriormente, el punto de encuentro común de todas estas funcionalidades es el uso del IH como elemento centralizado para la gestión de la información, por lo que el único requisito necesario sería la accesibilidad a este elemento por parte de los modelos. Por ende, el módulo OT podría ser reemplazado por una herramienta *serverless cloud-native* (véase apartado 2.6.6) como OpenFaaS o Knative, en las que se podrían configurar las ejecuciones de funciones empaquetadas en contenedores junto con sus parámetros de entrada y salida como reacción a ciertos eventos emitidos por la plataforma, así como de elementos de visualización como Grafana. La propia herramienta *serverless* se encargaría de guardar los resultados de ejecución de las funciones en la base de datos Elasticsearch del IH.

Por último, si se quisiera ir un paso más allá con el propósito de adaptar la propia arquitectura de la plataforma a un modelo de diseño basado en el continuo de computación, se debería realizar un ejercicio de rediseño, sobre todo para intentar eliminar la dependencia de un elemento con un grado de centralización de la información tan elevado como es el IH. Una posible propuesta enfocada en cálculo del PEI consistiría en ejecutar los agentes en un dispositivo más cercano a la fuente de datos para almacenar los eKPIs en una base de datos temporal, ya que tampoco sería eficiente incluir una base de datos históricos en la capa *edge* de la arquitectura. Seguidamente, el modelo del PEI podría dividirse en microservicios para ejecutar cada segmento en su ámbito correspondiente, para finalmente ejecutar el elemento agregador en la capa más *cloud* de la arquitectura y guardar los resultados en el IH centralizado, permitiendo borrar los datos que se han utilizado para ejecutar este modelo. Al mismo tiempo esta división del modelo aumentaría la seguridad e integridad de los datos reales sobre la emisión de gases contaminantes y de los actores involucrados en las operaciones portuarias, puesto que estos datos se mantendrían cerca de su fuente al solamente enviarse índices compuestos a

elementos externos situados en capas superiores de la arquitectura. De esta manera, se conseguiría una reducción del tamaño de la base de datos central y un mejor aprovechamiento de los recursos que conforman el continuo de computación.

3.2.2. VALKNUT

El proyecto "VALKNUT (*VALue Knowledge UTility*): Plataforma Smart Terminal para optimización de operaciones logísticas portuarias" fue un proyecto de I+D+i financiado por la Agencia Estatal de Investigación dentro del programa "Retos-colaboración", cuyo objetivo era fomentar la cooperación y transferencia tecnológica entre organismos de investigación y empresas, que a su vez estaba enmarcado dentro del Plan Estatal de Investigación Científica y Técnica y de Innovación 2017-2020. La propuesta tecnológica del proyecto consistía en la creación de una plataforma IoT abierta e interoperable mediante diferentes servicios, propios o de terceros, que permitiera a las terminales portuarias disponer de un control completo en tiempo real de los parámetros que gobiernan dichos procesos y ofrecer una herramienta para la optimización del consumo temporal y de recursos gracias a la modelización, simulación y optimización matemática de sus procesos logísticos.

Como se puede observar, el objetivo del proyecto no dista demasiado del que se pretendía alcanzar en el proyecto PIXEL. Además, si se observa la arquitectura de bloques de la solución desarrollada en el proyecto VALKNUT se pueden encontrar varias similitudes. No obstante, a juicio del doctorando, existen un par de diferencias claras entre ambos proyectos: (i) el alcance de la solución, puesto que VALKNUT se trataba de un proyecto de investigación nacional con un caso de uso concreto de aplicación inmediata y con unos recursos más limitados; y (ii) se pretendía desarrollar un solo algoritmo de optimización enfocado a las operaciones y procesos de logística portuaria en las terminales.

En cuanto a la implementación de esta solución, dadas las características de este proyecto se optó por desplegar elementos pertenecientes a la plataforma IoT FIWARE con algunas adaptaciones necesarias, en lugar de desarrollar una plataforma IoT a más bajo nivel como en el caso de PIXEL. Debido a que era necesario recopilar principalmente datos de sensores situados en las grúas de las terminales, además de otro tipo de información como la planificación de operaciones en la terminal, se decidió seguir el modelo de agentes definido por PIXEL. Esta era una decisión bastante obvia, puesto que también se utilizaba FIWARE Orion como gestor de contexto a modo de puerta de entrada de la información en la plataforma.

En este caso, el elemento más innovador de la plataforma VAKLNUT se trataba claramente del algoritmo de optimización de dos operaciones diferentes: (i) **planificador de atraques**, proceso en el que las grúas STS (*ship-to-shore*) descargan contenedores de los barcos en la terminal; (ii) **planificador de patio**, proceso en el que se colocan de manera organizada los contenedores descargados en la terminal. Este algoritmo se nutría de los datos introducidos en la plataforma

IoT, así como de ciertos elementos de configuración introducidos por los operadores. Asimismo, tanto su gestión operacional como la visualización de resultados estaba disponible a través de una aplicación web que actuaba como una suerte de panel de control de la plataforma.

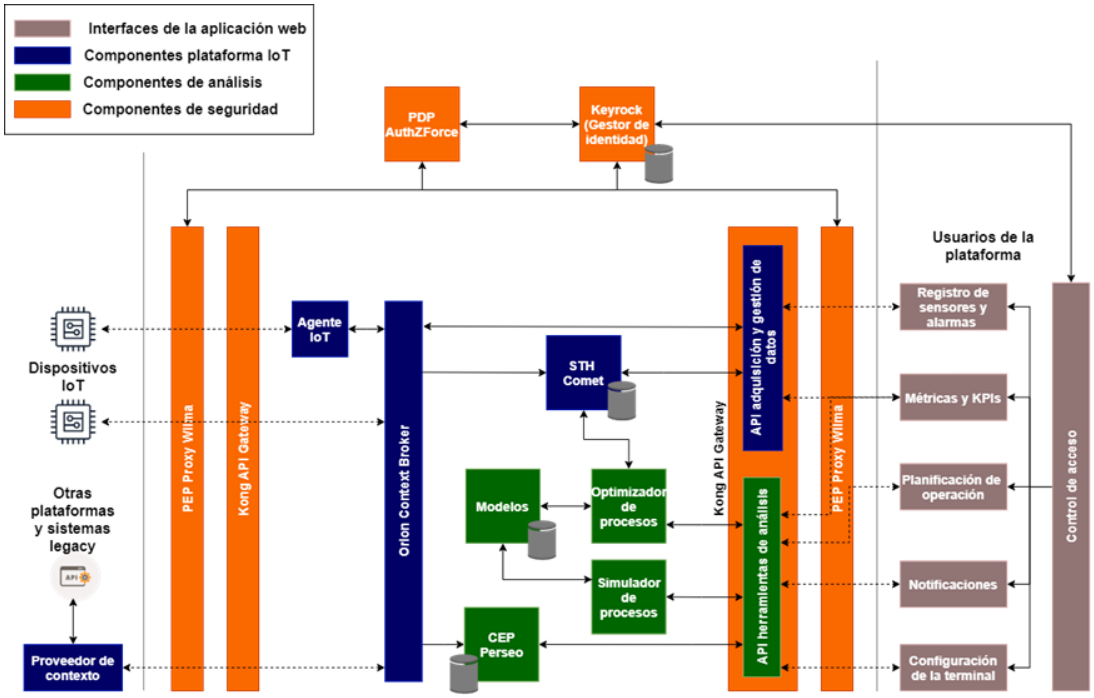


Figura 3.4: Arquitectura de la plataforma IoT de Valknut

Finalmente, para el proyecto VALKNUT aplicaría la misma reflexión que se ha desarrollado para el proyecto PIXEL, puesto que ambas utilizan contenedores como tecnología base de despliegue. Sin embargo, dada la naturaleza centralizada de VALKNUT y con un ámbito de aplicación tan concreto, no se atisba la necesidad de extrapolarla a un entorno como el continuo de computación.

3.3. Sistemas descentralizados *Cloud-Edge*

El avance del *cloud computing* alcanzó en pocos años un alto nivel de adopción y madurez, como se puede comprobar en la evolución de Kubernetes junto con otros grandes proyectos vinculados a su ecosistema. Esta acción estuvo liderada por la CNCF y enmarcada en una acción fuertemente vinculada a un espíritu de código abierto, como se puede comprobar en las licencias de estas tecnologías. No obstante, estos proyectos estaban realmente impulsados por las grandes empresas tecnológicas estadounidenses, puesto que veían el desarrollo de estas tecnologías como una manera de que impulsar sus propias plataformas de computación en la nube. De esta manera, acabaron convirtiéndose en dominadores del sector, siendo ejemplos predominantes AWS, Microsoft Azure o Google Cloud. K8s es un claro reflejo de este hecho, ya que, aunque se trata de un proyecto colaborativo y libre, los principales actores de la computación en la nube han acabado adaptándolo

mediante la creación de nuevas distribuciones o directamente ofreciéndolo como una mera extensión de sus servicios.

Esta consolidación convirtió a la computación en la nube en una tecnología ampliamente utilizada y fiable, reduciendo la necesidad de seguir desarrollando proyectos de investigación a nivel europeo enfocados exclusivamente en este ámbito, pero que al mismo tiempo abrió la puerta a nuevos desafíos como la orquestación inteligente de servicios o cargas de trabajos aprovechando estos entornos más uniformes. De manera similar, las plataformas IoT centralizadas también lograron un grado significativo de desarrollo y estandarización, logrando cubrir muchas de las necesidades iniciales de interoperabilidad e incluso integrándose en las plataformas dominantes en la nube que han sido anteriormente mencionadas.

Sin embargo, el crecimiento exponencial de los dispositivos IoT junto con el volumen masivo de datos generados por éstos, plantearon nuevas limitaciones y desafíos para estas arquitecturas centralizadas. En consecuencia, los modelos basados exclusivamente en la nube comenzaron a mostrar deficiencias en aspectos como la latencia, el uso eficiente del ancho de banda y el procesamiento en tiempo real de los datos, especialmente en aplicaciones en las que el tiempo de respuesta es un factor crítico [4]. Por este motivo, nació el paradigma del *edge computing* con el propósito de adaptarse a esta nueva realidad e intentar dar una solución a estos retos. Asimismo, la EC puso un gran interés en este nuevo paradigma, puesto que se atisbaba la oportunidad de establecer la tecnología europea como referente, y así liderar las iniciativas de investigación y de transferencia tecnológica en un sector claramente emergente. Además, el hecho de que el sector tecnológico europeo contara con una amplia experiencia en la implementación de sistemas basados en el IoT fue visto como una ventaja estratégica en el desarrollo de un paradigma de computación diseñado para interactuar estrechamente con el ecosistema IoT. De esta manera surgió la iniciativa NGIoT (Next Generation IoT) [241], en la que se englobaron diversos proyectos de investigación europeos financiados a través del programa H2020. El objetivo principal de esta iniciativa fue impulsar el desarrollo de una nueva generación de tecnología IoT, integrándola en arquitecturas descentralizadas que aprovecharan la colaboración entre los niveles *cloud* y *edge*.

Figura 3.5: Proyectos de investigación dentro de la iniciativa NGIoT

Estas iniciativas comparten una arquitectura descentralizada en las que el peso de las capas *edge* e IoT van ganando protagonismo, pero aun manteniendo un alto nivel de dependencia de la capa superior, que corresponde a la parte *cloud* o

más centralizada. Es por este motivo por el que aún no se puede hablar de un continuo de computación unificado y más distribuido, transición que se producirá en iniciativas futuras. Por último, esta introducción no estaría completa sin destacar el papel fundamental que, a partir de este momento, desempeñarán los sistemas de orquestación inteligentes. Estos sistemas operan sobre tecnologías de orquestación o de gestión ya consolidadas (siendo K8s otra vez el ejemplo más ilustrativo) para desplegar servicios de manera automatizada y eficiente en respuesta a una serie de requisitos especificados por los usuarios.

3.3.1. ASSIST-IoT

ASSIST-IoT (*Architecture for Scalable, Self-*, human-centric, Intelligent, Secure, and Tactile next generation IoT*) fue un proyecto financiado por la Comisión Europea dentro del programa Horizonte 2020, cuya duración comprendió el período entre noviembre de 2020 y marzo de 2024 [242]. Su objetivo principal consistió en el desarrollo de una arquitectura de referencia abierta y descentralizada para integrar el IoT de nueva generación. Esta arquitectura fue diseñada para establecer una clara separación entre los planos de datos y de control, además de integrar tecnologías emergentes como la inteligencia artificial, el aprendizaje automático, realidad aumentada y blockchain. ASSIST-IoT puso su foco en la habilitación de aplicaciones centradas en el ser humano (*human-centric*) en diversos sectores verticales, de modo que la validación de esta arquitectura se realizó en pilotos sobre: (i) automatización de puertos marítimos, (ii) mejora de la seguridad de los trabajadores involucrados en la construcción de edificios, y (iii) monitorización y diagnósticos de problemas en flotas de vehículos terrestres.

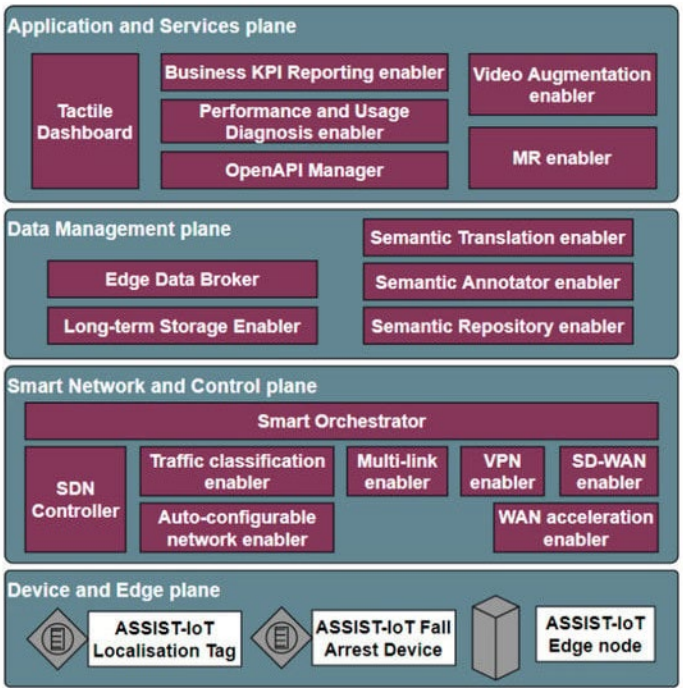


Figura 3.6: Arquitectura de bloques de ASSIST-IoT [243]

La arquitectura de bloques de ASSIST-IoT se divide en el desarrollo de varios componentes virtualizados llamados *enablers* que pertenecen a un ámbito de aplicación determinado, como se muestra en la Figura 3.6: (i) Plano de aplicación y servicios, (ii) Plano de gestión de datos, (iii) Plano de red inteligente y de control, y (iv) Plano de dispositivos y del *edge*. La colección completa de *enablers* clasificada por su ámbito de aplicación se puede consultar en el siguiente esquema [243].

Desde un primer momento, los líderes técnicos del proyecto decidieron abrazar los principios de diseño *cloud-native* y el uso de tecnologías bajo el paraguas de la CNCF como muestra de su ambición. Esta decisión se plasmó en la asunción de clústeres de K8s como punto de partida en términos de infraestructura en lugar del uso de contenedores aislados con Docker, tal y como ocurrió en los proyectos descritos en el anterior apartado. De igual manera, esta decisión conllevó el uso de tecnologías relacionadas como Cilium, a modo de CNI de los clústeres y como herramienta habilitadora tanto de la conexión entre clústeres y servicios (*service mesh*) como de políticas de red. Además, se acordó el uso de Helm charts como tecnología de empaquetado de los diferentes *enablers* que se desarrollarían durante el proyecto.

Como se ha comentado anteriormente, la piedra angular sobre la que se construye la arquitectura de ASSIST-IoT son los llamados *enablers*. La definición canónica de *enabler* según la documentación oficial del proyecto es "una colección de componentes de *software* (y posiblemente *hardware*) —que se ejecutan en nodos de computación— que trabajan conjuntamente para proporcionar una funcionalidad específica del sistema". Entrando en detalles técnicos, un *enabler* está compuesto internamente por un conjunto de componentes desplegados como contenedores, los cuales pueden comunicarse entre sí para aportar una funcionalidad determinada. Sin embargo, estos componentes están realmente aislados del entorno exterior, puesto que un *enabler* solo es accesible a través de una serie de interfaces predefinidas. Este diseño refleja claramente el concepto de encapsulación, dado que, dentro del marco de ASSIST-IoT, un *enabler* se considera una unidad indivisible que proporciona interfaces de comunicación bien definidas y una funcionalidad concreta y autónoma.

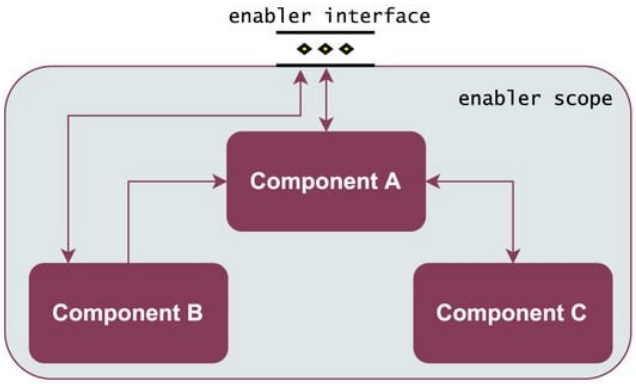


Figura 3.7: Arquitectura de bloques de un enabler de ASSIST-IoT [243]

El primer paso para cumplir con el concepto de encapsulación definido era la creación de una estrategia de empaquetado común para todos los *enablers*. Después de acordar el uso de Helm como herramienta de empaquetado descartando otras alternativas pujantes en ese momento como Kustomize o Juju, el doctorando lideró esta iniciativa de empaquetamiento dentro del proyecto. Ésta se inició con la definición de una serie de *labels* de K8s comunes para todos los *enablers* de manera que sirvieran para identificar de manera única e inequívoca (i) el *enabler* en sí, (ii) la instancia del *enabler* para diferenciar entre diferentes despliegues de un mismo *enabler*, y (iii) los componentes que forman parte de un *enabler* con sus permisos de exposición al exterior [244]. Asimismo, estas *labels* servían a su vez como elemento habilitador de la encapsulación de las comunicaciones con el exterior de los *enablers*, puesto que, aunque existen diversas soluciones para lograr esta funcionalidad, en ASSIST-IoT se eligió aprovechar las funcionalidades de políticas de red proporcionadas por Cilium.

Una vez definidas estas etiquetas, el siguiente paso natural era la creación de los Helm charts. Sin embargo, dado que se trataba de una tecnología de reciente desarrollo, la mayoría de los desarrolladores no contaba con experiencia previa en su uso, dando lugar a un posible riesgo de ruptura en la cadena de desarrollo del proyecto. Esta brecha se intentó solucionar mediante el desarrollo una herramienta que permitía crear un Helm chart desde cero a partir de las respuestas de los usuarios a una serie de preguntar sobre el *enabler* que se deseaba empaquetar (número de componentes, imagen de contenedor, variables de entorno, etc.). Esta herramienta fue desarrollada por el doctorando, de manera que sirvió para consolidar sus conocimientos sobre el empaquetado aplicaciones usando métodos *cloud-native* y, además, también se convertiría en el embrión del *Helm chart generator* que será la pieza clave en la estrategia de empaquetado de los desarrollos realizados durante esta tesis doctoral.

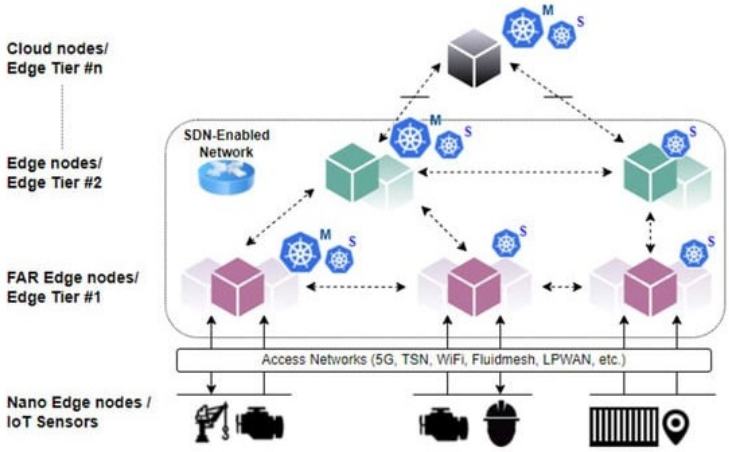


Figura 3.8: Arquitectura de la infraestructura de ASSIST-IoT en forma de clústeres de K8s [244]

Volviendo a la arquitectura general en cuanto a despliegue e infraestructura del proyecto, ésta estaba formada por diversos clústeres de K8s (así que es una

condición indispensable que todos los nodos de computación formen parte de un clúster de K8s) desplegados en diferentes regiones formando una topología de tipo árbol jerárquico, de manera que cada clúster debía de poder establecer comunicaciones con un clúster superior o más centralizado. En última instancia, todos los nodos dependen de un único clúster central que actúa como elemento de orquestación común. Para los nodos con recursos de computación limitados, se eligió utilizar la distribución K3s, puesto que está enfocada para ser instalada en este tipo de nodos (véase sección 2.6.5.2).

A partir de esta definición de la arquitectura, se empezó a diseñar el *enabler* más importante del proyecto: el *Smart Orchestrator*, cuyo propósito obedecía a los requisitos de creación de sistemas de orquestación y despliegue de aplicaciones inteligentes descrito en la introducción a esta sección. En su primera etapa de desarrollo, se exploró el uso de OSM (*Open Source MANO*) debido a que permitía la gestión de clústeres de K8s remotos para manejar el despliegue de Helm charts (y por tanto de *enablers*) *en* ellos. El único requisito era que la API del clúster fuera accesible por OSM, de manera que en muchos casos era necesario exponerla en IPs públicas, una práctica que no está recomendada. En cualquier caso, el mero uso de esta herramienta no permitía realizar el despliegue de *enablers* de manera inteligente u obedeciendo a ciertas políticas. Por ello, se combinó OSM con la herramienta *mck8s*, que proporcionaba una funcionalidad clave: la asignación automática de servicios a nodos de acuerdo con unas políticas de orquestación previamente definidas. De esta manera, se añadió a los Helm charts de los *enablers* la posibilidad de especificar parámetros relacionados con el uso previsto y el límite de recursos de computación de cada componente interno.

El diseño anterior no tardó en mostrar sus limitaciones, especialmente en entornos *edge* más dinámicos. Mientras que esta herramienta de orquestación funcionaba en entornos más controlados donde los clústeres podían exponerse usando IPs públicas y estáticas, como en entornos controlados con elementos de computación estáticos o redes corporativas privadas, y su número era reducido, ocurría todo lo contrario en otro tipo de casos de uso. Este el caso del piloto del proyecto titulado "Monitorización cohesiva de vehículos", cuyo propósito era la monitorización de diversos parámetros (emisiones de gases contaminantes, etc.) de una flota de vehículos durante su funcionamiento, es decir, con los vehículos en movimiento. Además, estos vehículos contaban con un GWEN (*Gateway/Edge Node*, un SBC diseñado y ensamblado durante el proyecto [245]) instalado y conectado a la red mediante una conexión 4G, deshabilitando por tanto la posibilidad de configurar direcciones IPs públicas y estáticas alcanzables por el orquestador. Por consiguiente, un objetivo evidente era el despliegue de ciertos *enablers* en la GWEN de los vehículos utilizando el módulo Smart Orchestrator, ya que esta actúa como nodo de computación del vehículo. Este escenario muestra claramente un entorno en el que era necesario realizar un diseño de la arquitectura de la solución que incluyera parámetros *edge-native* [246].

La primera decisión consistió en descartar el uso de OSM, puesto que fue diseñado para adecuarse a entornos puramente *cloud* en el ámbito de las NFVs y no acaba de encajar en entornos *Cloud-Edge*. En su lugar, se optó por sustituir este componente por un desarrollo propio que consistía en un motor de despliegue y gestión de Helm charts en los clústeres de K8s. Además, se extendió el módulo de asignación con la utilización de la herramienta de predicción de IA NeuralProphet, y también se incorporó el *context broker* Orion como elemento de monitorización de los clústeres. De todos modos, la mayor modificación se enfocó en el cambio de paradigma de despliegue, evolucionando de un modo de despliegue basado en comunicaciones síncronas 1:1 (un *enabler* desplegado en un único clúster) a un modelo 1:N (varias instancias de un mismo *enabler* desplegadas en una serie de clústeres), basado en comunicaciones asíncronas mediante el uso del *broker* de mensajería del *enabler* EDBE (Edge Data Broker Enabler, una extensión del *broker* MQTT VerneMQ). Gracias a este nuevo enfoque, los clústeres K3s situados en elementos finales de la arquitectura, como lo son las GWEN instaladas en los coches, no necesitan contar con una IP estática y directamente accesible por el orquestador, ni tampoco ser configurados de manera individual en el orquestador, ya que simplemente deben subscribirse a un *topic* determinado disponible en una dirección públicamente accesible y conocida. Asimismo, este rediseño implicó la incorporación de un agente en cada clúster *edge*, que es el elemento que realmente gestiona el ciclo de vida de los *enablers* en el clúster reaccionando a los eventos recibidos en el EDBE, permitiendo un modo de funcionamiento más flexible, dinámico y escalable [246].

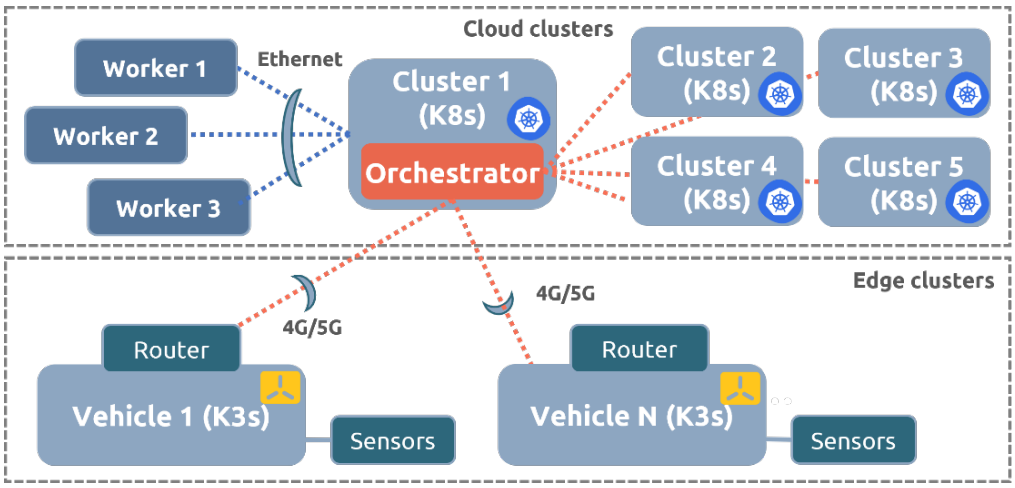


Figura 3.9: Arquitectura de bloques del sistema de orquestación de ASSIST-IoT

Propuesta de mejoras hacia el paradigma del continuo Cloud-Edge-IoT

El proyecto ASSIST-IoT priorizó desde el comienzo la utilización de tecnologías *cloud-native*, como son K8s o Cilium, para cimentar el desarrollo de su solución sobre una base formada por tecnologías emergentes y con gran potencial, aunque esta decisión supusiera una pronunciada curva de aprendizaje para su

equipo técnico. Este esfuerzo fue recompensado vistos los resultados obtenidos, que tuvieron un gran impacto dentro de la iniciativa NGIoT e inspiraron la propuesta de futuros proyectos como aerOS, que se describirá en la siguiente sección. Sin embargo, aunque en su momento fue realmente inspirador e innovador, con el paso de los años se pueden apuntar diversas mejoras dada la evolución exponencial que ha experimentado el *edge computing*.

En primer lugar, se puede apuntar que la arquitectura de implementación de ASSIST-IoT cuenta con una dependencia excesiva de K8s, puesto que como se ha indicado previamente todos sus nodos deben formar parte de un clúster. Este hecho es especialmente limitante en el caso de nodos con capacidades de computación limitadas, típicamente desplegados en el nivel *edge* de la arquitectura. Aunque se apostó por la utilización de distribuciones de Kubernetes con un consumo de recursos más reducido como son K3s o MicroK8s, el despliegue de un clúster de un solo nodo en elementos finales de la arquitectura (p. ej. GWEN instalada en un vehículo) puede ser visto como un uso ineficiente de los recursos de computación. Esto es debido a que parte de estos recursos se dedica a ejecutar el plano de control del clúster en lugar de ser destinado de manera exclusiva a la ejecución de cargas de trabajo. De manera paralela al proyecto empezaron a desarrollarse sistemas de orquestación de contenedores específicamente diseñados para estos casos de uso, como pueden ser KubeEdge o Baetyl, en los que generalmente es suficiente con instalar un pequeño agente en los nodos *edge*, de manera que el plano de control se mantiene en los nodos más potentes. No sería necesario aportar aquí más detalles sobre estos nuevos sistemas de orquestación, ya que en el apartado 2.6.5.2 se ha realizado un estudio extenso sobre la materia.

La siguiente propuesta de mejora está enfocada exclusivamente al *Smart Orchestrator enabler*. El modelo de topología utilizado por este orquestador responde a un modelo centralizado de tipo árbol jerárquico, en el que el proceso de orquestación está totalmente controlado por un único elemento central desplegado en el clúster considerado como *cloud* o de más alto nivel. Por ende, este nodo central se trata de un potencial candidato para convertirse en un punto de fallo crítico, inhabilitando el proceso de despliegue de *enablers* en los demás clústeres del sistema si éste o su *broker* de comunicaciones (EBDE) falla. Además, la información sobre el estado de los nodos y los clústeres a los que pertenecen los *enablers* está totalmente centralizada en el nodo principal, aunque también se encuentra de manera local y aislada en cada uno de ellos. De esta manera, no existen mecanismos de federación o de compartición de este tipo de información entre los elementos de la arquitectura. El establecimiento de herramientas colaborativas entre diferentes elementos de la arquitectura podría ser de gran ayuda en la implementación de una posible distribución de los componentes del *Smart Orchestrator*.

Como punto a favor del *Smart Orchestrator*, su diseño final atisbaba la futura separación de responsabilidades en los modelos de orquestación modernos [246]. En resumen, se divide el proceso de orquestación en dos niveles: el proceso de selección inteligente del nodo de despliegue, que se produce en nodos de computación más

potentes; y el proceso de despliegue del servicio en el nodo elegido, que es gestionado por un agente instalado en el mismo nodo. Este concepto se desarrollará con todo tipo de detalles en la siguiente sección debido a que es uno de los pilares fundamentales sobre los que se construye el modelo de orquestación de servicios propuesto en esta tesis doctoral.

Respecto al proceso de comunicación entre el orquestador y los diferentes nodos, el uso de un *broker* de comunicaciones basado en MQTT no sería de igual manera óptimo para un entorno de comunicaciones *Cloud-Edge*, puesto que esta tecnología fue diseñada para establecer comunicaciones en el ámbito del IoT. En los últimos tiempos han emergido soluciones como NATS, que mantienen la ligereza de los *brokers* MQTT, pero al mismo tiempo añaden funcionalidades específicas para su uso en este tipo de arquitecturas (baja latencia, arquitectura distribuida, etc.), como se ha apuntado en el punto 2.4.2.

3.4. Continuo de computación *Cloud-Edge-IoT*

El paradigma del *edge computing* siguió evolucionando durante los siguientes años, en los que, dejando de un lado las iniciativas de investigación mencionadas, se consolidó la tendencia de trasladar conceptos y tecnologías puramente *cloud-native* al *edge computing* para establecer el patrón de diseño *edge-native*. En este ámbito se engloba la creación de *container runtimes* específicamente diseñados para ejecutarse en máquinas con pocos recursos de computación, de manera que la ejecución de contenedores en el *edge* no sea un problema. También es una muestra de esta tendencia el desarrollo de soluciones de orquestación de contenedores enfocadas a la colaboración *Cloud-Edge*, ya sea mediante adaptaciones directas de K8s (KubeEdge) o mediante rediseños (Baetyl); o la creación de nuevas tecnologías habilitadoras de comunicaciones como *brokers* de mensajería (NATS) o *frameworks* de conexión de red (*libp2p*), tal y como se ha ido describiendo durante el estado del arte (capítulo 2).

Todo este tipo de iniciativas empezaron a converger en el concepto del continuo de computación *Cloud-Edge-IoT*, que engloba todos los niveles del espectro de computación (*cloud, edge, far-edge,* IoT...*)*, ya que sin ellas habría sido imposible concebir este concepto unificador. El propósito central de este continuo es lograr un cierto nivel de homogeneidad en las operaciones relacionadas con el control de recursos de procesamiento, así como en la orquestación e interconexión de aplicaciones desplegadas sobre estos. Asimismo, intenta difuminar la separación entre las capas del continuo, eliminando la necesidad de clasificar rígidamente los servicios o la infraestructura. De este modo, el despliegue de cada servicio se realiza únicamente teniendo en cuenta sus requisitos, e incluso este enfoque puede llegar a permitir que los usuarios finales puedan ser abstraídos hasta cierto punto de toda esta complejidad subyacente.

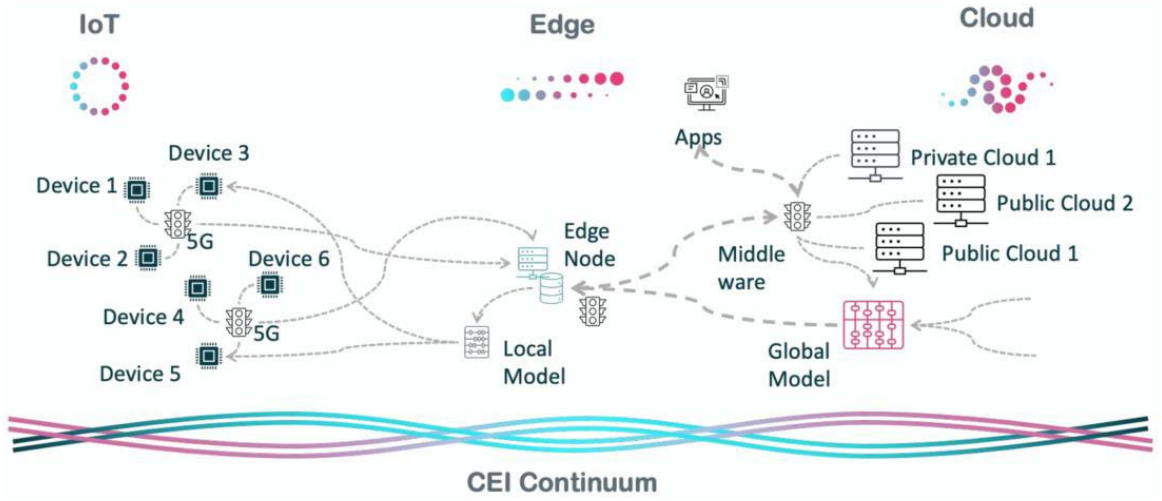


Figura 3.10: Continuo de computación *Cloud-Edge-IoT* según la iniciativa EUCEI [24]

Al igual que sucedió en el momento que emergió el *edge computing*, la EC calificó el continuo de computación como uno de sus objetivos clave en materia de innovación [247]. No obstante, la competencia en este campo es de igual manera elevada, puesto que aparte de competir con las compañías norteamericanas que cuentan con amplia experiencia en el sector, iniciativas cada vez más potentes y prometedoras relacionadas con el *edge* están siendo lideradas por actores asiáticos. En gran parte motivada por los buenos resultados que dieron acciones previas como la anteriormente descrita NGIoT, la respuesta por parte de la EC fue la creación de la iniciativa EUCloudEdgeIoT (EUCEI) con la finalidad de crear un foro común que englobara todas las acciones de investigación europeas relacionadas con el continuo de computación [24]. Este foro busca fomentar relaciones de sinergia y colaboración entre ellas, evitando que actúen como iniciativas aisladas, además de garantizar que estén alineadas con los objetivos y las metas a largo plazo de la Unión Europea (UE).

La canalización práctica de esta iniciativa fue estructurada en diferentes grupos de trabajo (llamados *Task Forces* —TFs), cada uno enfocado en un área específica de interés, y liderados por una entidad perteneciente a alguna de las acciones de investigación que la componen. Los grupos de trabajo son los siguientes: TF1 — Enlaces estratégicos, TF2 — Compromiso con el código abierto, TF3 — Arquitectura, TF4 — Compromiso con el ecosistema, TF5 — Mercado y sectores y TF6 — Comunicaciones.

Para el propósito que persigue esta tesis doctoral, la TF3 de arquitectura es sin lugar a duda la más interesante puesto que en ella se realizan discusiones y propuestas periódicas sobre las arquitecturas globales de los procesos más importantes del continuo de computación. Además, el doctorando ha podido ser parte activa de ella gracias a formar parte del equipo investigador del proyecto aerOS, en el que el grupo SATRD ejerce como coordinador.

La principal actividad actual de esta TF3 consiste en avanzar en la creación de un estándar que defina el diseño y el despliegue del continuo de computación.

El objetivo es establecer directrices conceptuales, tecnológicas, metodológicas y de adopción para el continuo, basándose en la experiencia y los casos de éxito de los proyectos RIA (*Research and Innovation Action*) financiados por la Comisión Europea. La concepción de este estándar se está abordando como un esfuerzo colectivo y colaborativo, en el que representantes de los distintos proyectos proponen ideas basadas en los desarrollos realizados en sus iniciativas. Estas propuestas son posteriormente debatidas y consensuadas entre los participantes, promoviendo un enfoque integrador que permita la convergencia de las diferentes perspectivas y avances tecnológicos. Finalmente, cuando concluya esta actividad, se pretende que sus resultados sean transferidos al programa de estandarización **ISO/IEC JTC 1/SC 41** [248] (estandarización en el área del IoT y de los gemelos digitales — *digital twins*) para que sea registrado como un estándar de la ISO (International Organization for Standardization). Actualmente, parte de la propuesta de esta tesis se encuentra contenida en dicha iniciativa, que ha sido aceptada ya como un ítem de trabajo preliminar (Preliminary Work Item — PWI) por dicha entidad de estandarización.

La TF3 está dividida a su vez en diferentes grupos de trabajo (en este caso llamados *Working Groups* — WGs) que definen los conceptos y bloques tecnológicos que conforman los procesos claves del continuo de computación. Estos WGs son: WG-1 Seguridad y Privacidad, WG-2 Fiabilidad y Reputación, WG-3 Gestión de los datos, WG-4 Gestión de los recursos, WG-5 Orquestación, WG-6 Red, WG-7 Monitorización y Observabilidad, y WG-8 Inteligencia Artificial [249].

El proyecto aerOS está liderando el WG-5, que se centra en definir el proceso de la orquestación inteligente de servicios o aplicaciones en los diferentes recursos de computación que componen el continuo. Durante las reuniones periódicas de este grupo de trabajo, la visión de aerOS en materia de orquestación logró imponerse como referencia gracias a la calidad del trabajo previamente realizado durante el proyecto, que queda perfectamente plasmado en los dos entregables de arquitectura D2.6 y D2.7 [250]. En este contexto es necesario destacar que tanto el doctorando como los directores de esta tesis fueron los impulsores principales del diseño de los bloques fundamentales que conforman la arquitectura del proceso de orquestación, haciendo hincapié desde un primer momento en el papel fundamental que debe tener la federación entre diferentes sistemas, áreas o dominios en los sistemas descentralizados o distribuidos, tal y como lo es el continuo. Este enfoque permitió consolidar una base sólida para garantizar la interoperabilidad y la eficiencia en escenarios de computación heterogéneos.

Dado el valor e importancia de esta contribución, resulta fundamental proporcionar una descripción general de la arquitectura de bloques propuesta, ya que esto no solo aporta claridad al lector, sino que también sirve como una introducción más sólida al contexto de la orquestación de servicios en el continuo.

En primer lugar, el módulo de **descomposición de servicios** fue definido para ilustrar que los servicios o aplicaciones que pretenden desplegar los usuarios en el continuo no suelen tratarse de entidades monolíticas. Por el contrario, en

realidad son una abstracción unificadora compuesta por múltiples componentes o módulos más pequeños. Cada uno de estos módulos cumple una función específica y, en conjunto, forman el "puzle" que constituye la aplicación final a desplegar. En muchos sistemas de orquestación el proceso se realiza y repite individualmente para cada componente, pero sin perder de vista la funcionalidad global del servicio.

La arquitectura de bloques de la funcionalidad de orquestación de servicios inteligentes definida por este WG-5 está efectivamente soportada por un bloque de **federación** transversal que habilita el acceso a la información clave para poder llevar a cabo este proceso. Esta información realmente se encuentra disponible en un ámbito más local de cada dominio, área o sistema, pero gracias a este bloque puede ser consultada desde cualquier punto del continuo, ya que define sistemas de compartición de la información basados en la colaboración entre dichos dominios.

Una de las mayores novedades en cuanto a la arquitectura de los procesos de orquestación radica en su división en dos bloques con una separación de funcionalidades y responsabilidades claramente definidas. Esta estructura se inspira, en cierta medida, en la división implementada por los *container runtimes*, donde se distingue entre componentes de alto y de bajo nivel. Asimismo, esta nomenclatura refleja la especialización de cada bloque, lo que permite una gestión más eficiente y organizada del proceso de orquestación:

- El bloque de **orquestación de alto nivel** (High-Level Orchestrator — HLO) se encarga de gestionar las peticiones de orquestación enviadas por los usuarios, que incluyen una serie de indicaciones y requisitos específicos que deben ser considerados en el proceso. Estas indicaciones son analizadas junto con la información actualizada sobre el estado del continuo, que es recopilada y compartida gracias a la federación de la información, para, en última instancia, determinar en qué nodo de computación del continuo se realizará el despliegue del servicio. Para tomar esta decisión, el bloque cuenta con un módulo de inteligencia (usualmente soportado por modelos de IA) que se alimenta de los datos anteriores.
- El bloque de **orquestación de bajo nivel** (Low-Level Orchestrator — LLO) tiene como función materializar la decisión de despliegue tomada por el bloque de alto nivel. Este bloque tiene un carácter más funcional o de acción directa, puesto que no cuenta con una parte de inteligencia. Además, cuenta con un mayor grado de especificación debido a que el despliegue del servicio depende del motor de ejecución de aplicaciones del nodo seleccionado. Por ejemplo, si el servicio se despliega como un contenedor y el nodo utiliza Docker para ejecutar contenedores, el LLO debe interactuar con el Docker Engine para ejecutar el contenedor.

Por último, el módulo de **reorquestación** fue incluido para plasmar la naturaleza cambiante de un entorno tan heterogéneo como el continuo *Cloud-Edge-IoT*. En este tipo de entornos, pueden ocurrir innumerables eventos que

imposibiliten la ejecución de un servicio en el nodo originalmente seleccionado, como el apagado del nodo, una disminución de su nivel de confianza (o *trustworthiness*), la sobrecarga de sus recursos o su desconexión de la red. Por ende, se consideró indispensable desarrollar un mecanismo capaz de detectar estas situaciones y responder en consecuencia mediante el inicio de un proceso de reorquestación. Este proceso es llevado a cabo bajo los mismos parámetros en los que se produjo la orquestación original, pero se basa en datos actualizados del contexto del sistema, lo que garantiza que las decisiones tomadas sean óptimas y estén adaptadas a las condiciones actuales del continuo [249].

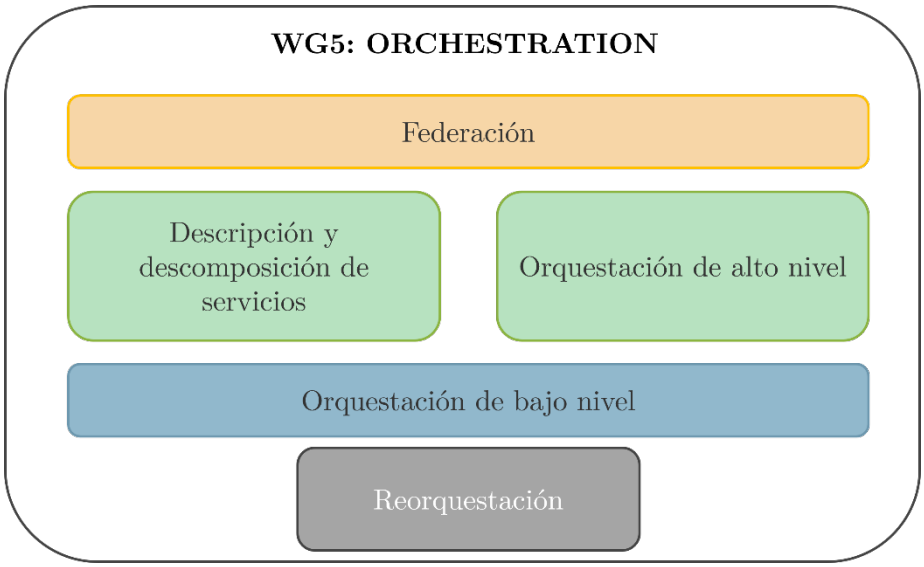


Figura 3.11: Bloques fundamentales del sistema de orquestación definido por el WG5 de la TF3 de EUCEI

Proyectos con el objetivo de desarrollar un Meta-OS para el continuo

Dentro de las RIAs (*Research and Innovation Action*) o iniciativas de investigación que impulsó la EC para abordar el desafío del continuo de computación *Cloud-Edge-IoT* destacó la acción HORIZON-CL4-2021-DATA-01-05 para abordar la creación de Meta-OS para gestionar el continuo [251]. En términos generales, un Meta-OS ofrece servicios y funcionalidades similares a las de un sistema operativo convencional, pero diseñado para operar en un entorno donde múltiples recursos heterogéneos y distribuidos están integrados de cierta forma uniforme, como puede ser el flujo de intercambio de datos o la cooperación de diferentes nodos. En el caso del continuo de computación impulsado por la EC, un Meta-OS se define como una plataforma de *software* de código abierto cuyas funciones equivalen a las de un sistema operativo. Estas funciones incluyen la monitorización del estado del sistema, la abstracción del *hardware*, el manejo de datos e identidades, y la gestión de aplicaciones y paquetes.

Los proyectos financiados dentro de este *topic* (entre los que se encuentra aerOS) tienen un objetivo compartido, que se trata claramente de la definición y desarrollo de un Meta-OS para la gestión del continuo. Sin embargo, cada uno se centra en un aspecto más en concreto debido a los múltiples desafíos que aún están por resolver en este ámbito, además de aportar su visión única del concepto de continuo de computación y de las funcionalidades de su Meta-OS. Gracias a la iniciativa EUCloudEdgeIoT, el intercambio de estos diferentes puntos de vista y enfoques permitirá enriquecer la solución final tanto a nivel individual como proyecto como a nivel global en las iniciativas de estandarización descritas anteriormente.

Finalmente, la propia iniciativa EUCloudEdgeIoT realizó un esquema comparativo entre los diferentes proyectos del clúster de MetaOS que se reproduce en la Figura 3.12. Los dos ejes comparativos de este esquema son el aspecto concreto del continuo que se pretende abordar y el contexto operacional en el que se engloba el proyecto. Por ejemplo, se puede observar como aerOS se centra especialmente en el *edge* operacional (operaciones cercanas a las fuentes de datos, sistemas distribuidos...) en comparación con otros proyectos que ponen su foco de atención en otros aspectos más relacionados con el *edge* para la infraestructura de operadores de telecomunicaciones (como la integración de comunicaciones 5G en el continuo), o la extensión del *cloud* para mejorar su integración en el continuo de computación.

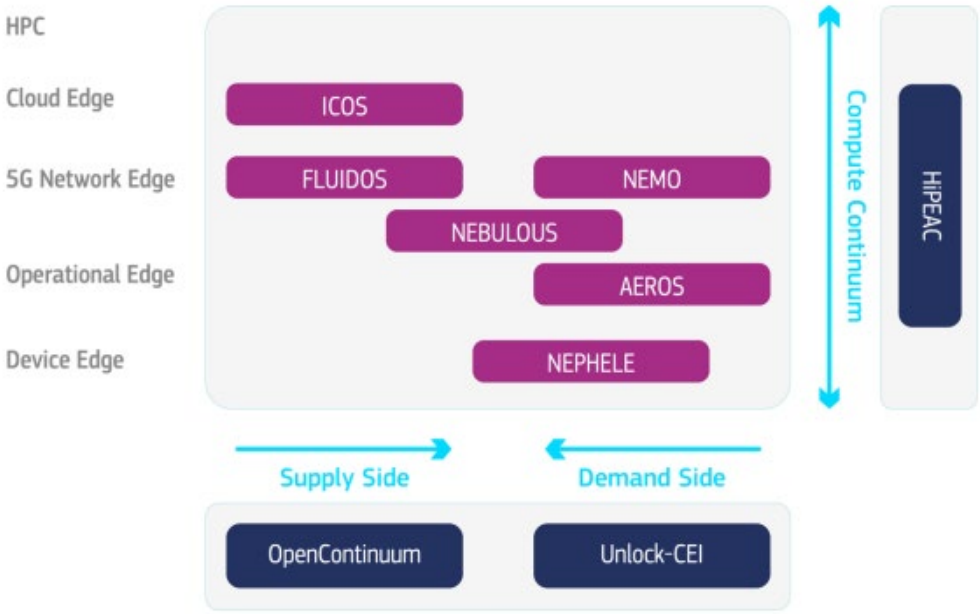


Figura 3.12: Esquema comparativo entre los proyectos del clúster de Meta-OS [24]

3.4.1. FLUIDOS

El Meta-OS concebido por el proyecto FLUIDOS se ha basado desde un primer momento en la idea de reutilizar herramientas que hayan sido desarrolladas previamente y estén ampliamente establecidas en la industria [252]. La idea es que FLUIDOS se pueda desplegar en cualquier tipo de *hardware* que sea compatible con la instalación de un sistema operativo de propósito general. Por ende, FLUIDOS puede instalarse en una gran variedad de recursos de computación localizados a lo largo del continuo, tanto en dispositivos con recursos limitados, como sistemas embebidos (Raspberry Pis, NVDIA Jetson...), como en portátiles, ordenadores convencionales y servidores *cloud* potentes. Asimismo, otro pilar fundamental de FLUIDOS es la asunción de que se va a desplegar en entornos de *zero trust*, con las consideraciones de seguridad y funcionamiento que conlleva.

La gran apuesta de FLUIDOS consiste en asumir que todos los recursos que serán gestionados por su Meta-OS forman parte de un clúster de K8s, de manera que actúe como una gran capa unificadora. Esta asunción o enfoque está respaldado por la existencia de las múltiples distribuciones y adaptaciones de K8s con diferentes requisitos, propósitos, rendimiento o propiedades de escalabilidad, pero que en última instancia comparten una misma API y funcionamiento a alto nivel, como también se ha visto a lo largo de la sección 2.6.5. Esta es sin lugar a duda la mayor diferencia entre FLUIDOS y los demás proyectos del clúster de Meta-OS, como aerOS o ICOS. Además, FLUIDOS utiliza KubeEdge para tratar de incorporar a nodos con recursos más limitados en su sistema, manteniendo así su enfoque totalmente centrado en K8s.

Como K8s está, por defecto, enfocado a la creación de clústeres independientes entre ellos, FLUIDOS incorpora a su base tecnológica Liqo, que se trata de una herramienta de código abierto para establecer relaciones funcionales y colaborativas entre diferentes clústeres de K8s. De hecho, Liqo actúa en FLUIDOS como uno de los cimientos principales que permiten habilitar este Meta-OS para gobernar el continuo, puesto que está presente en las funcionalidades relacionadas con la gestión de la red, almacenamiento y el despliegue de servicios. De esta manera, los componentes de FLUIDOS se construyen sobre la pila tecnológica formada por la unión de clústeres de K8s mediante Liqo [253], ya sea extendiendo estas mismas tecnologías o desarrollando nuevas funcionalidades que interactúen directamente con ellas [254].

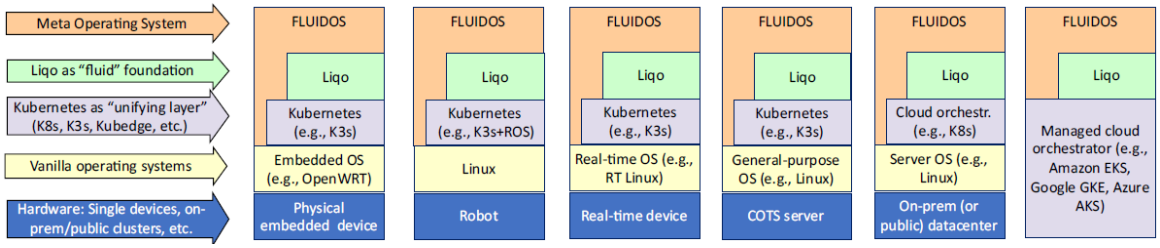


Figura 3.13: Tecnologías utilizadas por el Meta-OS de FLUIDOS [254]

El sistema de orquestación de servicios de FLUIDOS está basado en un nuevo protocolo que ha sido desarrollado específicamente para ser usado en su Meta-OS: el protocolo REAR (Resource Advertisement and Reservation) [255]. Los diferentes recursos de computación son representados en los llamados *flavors*, que son en primer lugar seleccionados por los usuarios para seguidamente realizar una reserva de recursos. Una vez se ha confirmado la asignación del *flavor* y la reserva de recursos adyacente, los usuarios pueden desplegar servicios y hacer uso de ellos.

Finalmente, en la siguiente estructura de bloques se pueden observar los componentes que componen el Meta-OS de FLUIDOS, su interacción entre ellos y su relación con las herramientas tecnológicas base (K8s y Liqo).

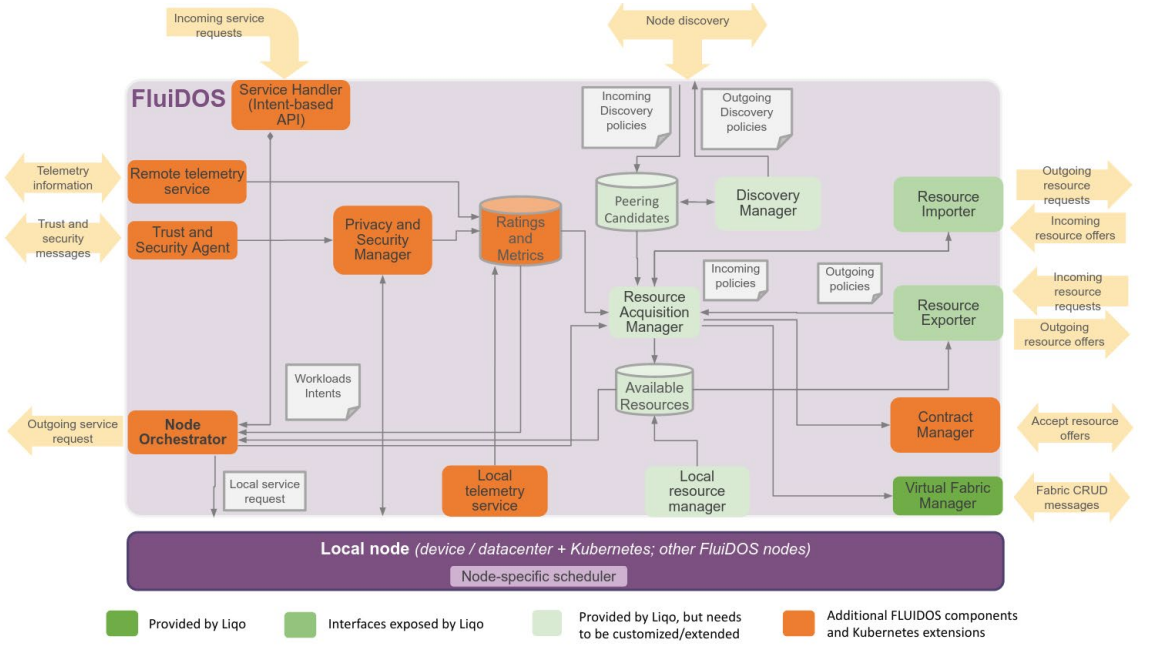


Figura 3.14: Arquitectura del Meta-OS de FLUIDOS [254]

3.4.2. ICOS

ICOS, resultado de otro de los proyectos del clúster, ha sido concebido como un Meta-OS distribuido a lo largo del continuo *Cloud-Edge-IoT*, diseñado para ser elástico, dinámico y móvil, permitiendo que los nodos se unan o abandonen el sistema de manera dinámica según sus necesidades, adaptándose fácilmente a posibles cambios en la topología del sistema [256]. Para cumplir con este propósito, ICOS define dos tipos principales de nodos [257]:

- **ICOS Controller**: es el responsable de gestionar el continuo, manteniendo un registro actualizado de la topología y disponibilidad del sistema. Además, gestiona el *runtime*, lanzando y supervisando la ejecución de servicios bajo demanda.

- **ICOS Agent**: es el encargado de ejecutar los servicios desplegados por los usuarios, gestionando la ejecución del código, el acceso a los datos, la telemetría y, eventualmente, la comunicación con otros Agents durante su ejecución. Los ICOS Agents están distribuidos a lo largo del continuo y abarcan desde dispositivos con capacidades de computación limitadas propios del *far-edge* hasta elementos con mayor potencia de computación propios de la nube.

Con estas definiciones, queda patente que el Meta-OS de ICOS ha sido diseñado para soportar una gran cantidad de nodos (en forma de *Agents*) heterogéneos y distribuidos a lo largo de una amplia área geográfica y del espectro del continuo. Para gestionar eficientemente este gran número de nodos y aprovechar las ventajas propias de los entornos *edge*, propone desplegar un conjunto de ICOS Controllers a lo largo del continuo, de manera que cubra todas las áreas posibles de forma equilibrada. De esta manera, los ICOS Controllers se organizan de manera horizontal y no estructurada, en la que cada uno gestiona un conjunto de Agents elegidos en base a criterios de proximidad. Esto da lugar a una arquitectura de control altamente distribuida, horizontal y sin una jerarquía claramente definida, profundizando en la naturaleza descentralizadora de su arquitectura.

Asimismo, el componente Lighthouse se ha diseñado para simplificar y automatizar los procesos dinámicos de incorporación, desconexión, reconexión y migración de los ICOS Agents hacia los ICOS Controllers. Este componente almacena datos y referencias a los *endpoints* necesarios para que los ICOS Agents y Controllers puedan establecer una relación de manera eficiente.

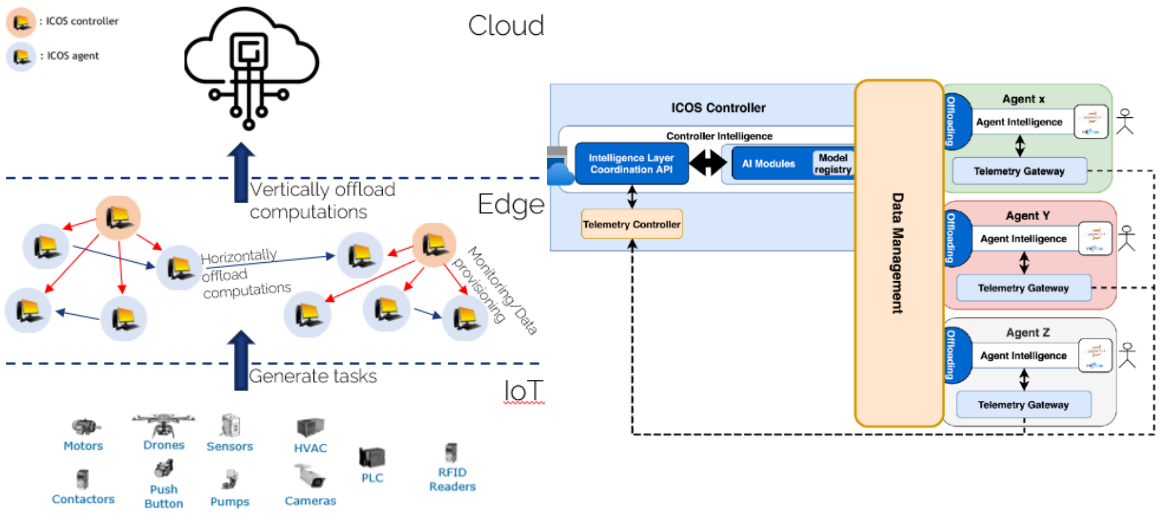


Figura 3.15: Arquitectura del Meta-OS del proyecto ICOS [257][258]

A nivel de arquitectura de bloques, el Meta-OS de ICOS se organiza en tres capas principales: Meta-Kernel, Seguridad e Inteligencia. La capa del Meta-Kernel es la encargada de proporcionar las principales funcionalidades propias de un sistema operativo al continuo, como lo son propia su gestión, la gestión del *runtime*,

así como el registro y telemetría. Esta capa está estrechamente integrada con la capa de seguridad, que tiene como responsabilidad garantizar la seguridad y la confianza en el sistema, y con la capa de inteligencia, que enriquece las acciones realizadas mediante enfoques innovadores basados en inteligencia artificial. Además, las capas de seguridad e inteligencia también interactúan entre sí para mejorar los análisis y acciones relacionadas con la seguridad utilizando modelos de IA avanzados. Por otro lado, actuando de manera transversal a estas capas principales, la capa de gestión de datos permite que los distintos componentes del sistema se comuniquen a través de servicios de datos seguros y distribuidos, mientras que el ICOS Shell proporciona a los usuarios acceso a todas las funcionalidades del sistema.

Estas tres capas principales se descomponen a su vez en múltiples componentes funcionales, cada uno con un conjunto claramente definido de responsabilidades e interfaces. Por lo tanto, cada componente implementa una o varias funcionalidades específicas del Meta-OS ICOS, permitiendo abordar la complejidad del sistema de manera más eficiente al fragmentarlo en bloques más pequeños y manejables. Esta división modular no solo facilita el análisis y diseño de cada bloque de forma independiente, sino que también optimiza su desarrollo e implementación final, promoviendo un enfoque más escalable, flexible y orientado a la mejora continua del sistema [258].

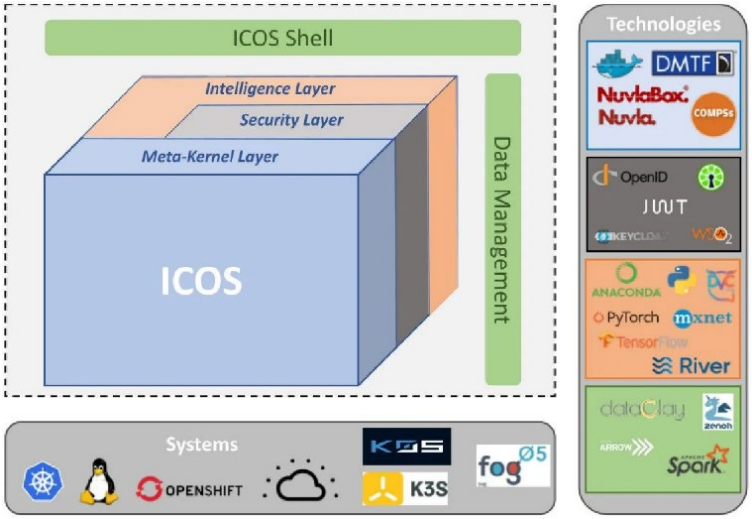


Figura 3.16: Componentes y tecnologías del Meta-OS ICOS [257]

3.4.3. aerOS

El principal propósito del proyecto aerOS es el diseño e implementación de un Meta-OS virtualizado y agnóstico a la plataforma en la que se despliegue para la gobernanza del continuo *Cloud-Edge-IoT* de una manera eficiente [8]. Por lo tanto, este Meta-OS ha de poder ser ejecutado en cualquier nodo de computación del continuo, denominado como *Infrastructure Element* (IE), siempre y cuando cumpla

con una serie de requisitos de funcionamiento, con independencia del *hardware* y sistema operativo que utilicen.

Por su parte, el Meta-OS de aerOS tiene dos objetivos principales a nivel tecnológico: (i) cubrir la brecha existente entre las tendencias e implementaciones actuales relacionadas con los espacios de datos; y (ii) plantear un nuevo paradigma para avanzar hacia una orquestación de datos, recursos de computación y servicios de usuarios más rápida y eficiente que abarque el espectro completo del continuo de computación. De esta manera aerOS pretende unificar los recursos pertenecientes tanto al *cloud* y al *edge computing* como al IoT [250].

Mientras que la descentralización de servicios y la segmentación de aplicaciones y servicios monolíticos han ido avanzando como conceptos de diseño, aerOS pretende dar un paso más allá al integrar los conceptos de gestión de recursos y orquestación de servicios en un ecosistema federado. En este contexto, aerOS ha diseñado y desarrollado un conjunto de servicios fundamentales que abordan la necesidad de abstraer los recursos individuales de cada elemento de computación y homogeneizar el acceso a sus capacidades de computación, al mismo tiempo que permite compartir y gestionar estos recursos de manera colectiva. Esto facilita una asignación eficiente de los recursos y una ubicación óptima de los servicios IoT, teniendo en cuenta sus restricciones y requisitos específicos.

La interconexión y comunicación de los servicios entre todos los dominios de aerOS constituyen los llamados *fabrics* (concepto que suele referirse a una arquitectura para sistemas distribuidos en la que diferentes elementos están interconectados y operan de una manera uniforme) del Meta-OS del proyecto (*network & compute fabric*, *service fabric* y *data fabric*), proporcionando la base sobre la que construir su sistema de orquestación descentralizada. Este sistema de orquestación sigue la arquitectura de bloques ilustrada en la Figura 3.11, puesto que, recordemos, aerOS está liderando el WG-5 de orquestación de la iniciativa EUCEI con la participación fundamental del doctorando.

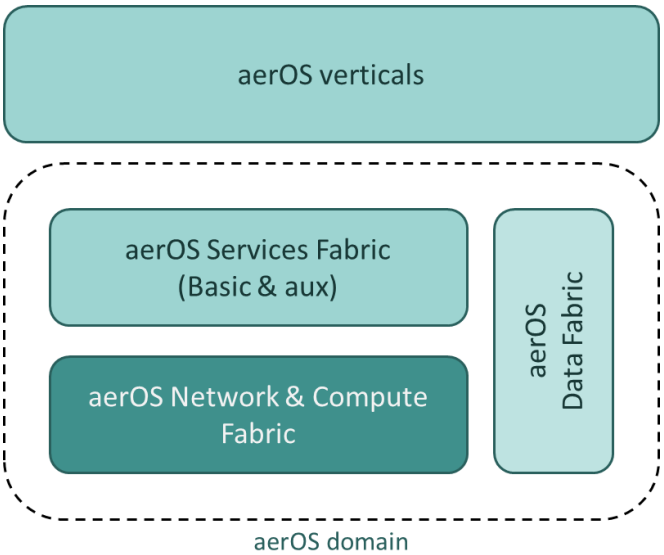


Figura 3.17: Diferentes *fabrics* presentes en los dominios de aerOS [250]

El Meta-OS de aerOS establece un requerimiento importante para la implementación de la federación entre dominios: todos los dominios deben contar con una dirección IP pública, o que al menos sea alcanzable (o enrutable) por los demás dominios del continuo. Al establecer este requerimiento, las comunicaciones entre los servicios básicos de cada dominio se realizan a través de una API *Gateway* que está expuesta a través de esta dirección pública de una manera segura.

De manera transversal, aerOS cuenta con un bloque llamado AAA (*Autentication, Authorization and Access*) para proveer de un control de acceso granular sobre las APIs y los datos expuestos; un bloque de reputación y fiabilidad (*Trustworthiness*) basado en la tecnología IOTA; y un conjunto de herramientas para el entrenamiento y despliegue de módulos de IA en el continuo de manera totalmente descentralizada, utilizando técnicas distribuidas como el llamado *Federated Learning* (FL).

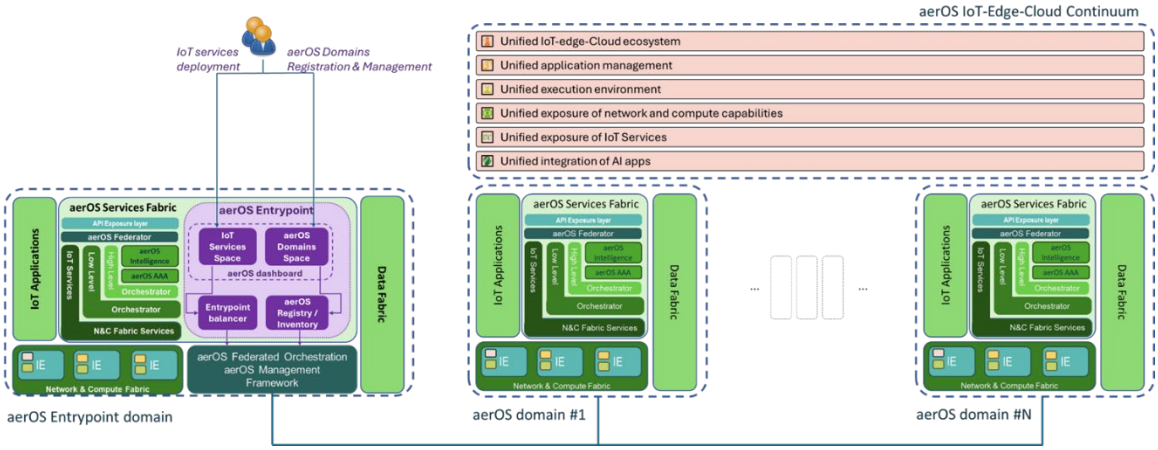


Figura 3.18: Arquitectura del Meta-OS de aerOS [250]

3.5.Comparación

A modo de conclusión para esta sección, se ha decidido crear una tabla comparativa de los diferentes proyectos que han aparecido durante la misma. En esta tabla se muestran elementos clave como su objetivo principal, su tipo de arquitectura, como implementan su solución para la orquestación de servicios y su factor diferencial. Además, esta tabla (Tabla 3.1) también ejerce como complemento a la Figura 3.1 en la que se presentaba la transición hacia el continuo de computación utilizando estos proyectos como ejemplo ilustrativo de los paradigmas de computación que representan.

Tabla 3.1: Comparación de proyectos de investigación descritos en la sección 3

Proyecto	Objetivo	Tipo de arquitectura	Orquestación de servicios	Monitorización	Factor diferencial
PIXEL	Creación de una plataforma IoT flexible para optimizar las operaciones en puertos marítimos de tamaño mediano y pequeño.	Plataforma IoT	No. Ejecución de modelos y algoritmos predictivos en forma de contenedores de manera centralizada.	No se contemplan diferentes nodos de computación distribuidos.	Plataforma IoT para alimentar modelos y algoritmos predictivos enfocados a los puertos marítimos y la reducción del impacto medioambiental
VALKNUT	Creación de una plataforma IoT para la optimización del consumo temporal y de recursos de los procesos logísticos de las terminales de contenedores portuarias	Plataforma IoT desplegada en una sola máquina	No	No se contemplan diferentes nodos de computación distribuidos.	Ejecución de un modelo de optimización de recursos en las terminales portuarias.
ASSIST-IoT	Desarrollo de un Meta-OS para la integración de los elementos heterogéneos del continuo.	Diseño descentralizado *Cloud-Edge* basado en una topología de árbol jerárquico en la que sus nodos son clústeres de K8s.	Orquestación centralizada e inteligente de servicios empaquetados como Helm charts: Smart Orchestrator *enabler*.	Monitorización centralizada del estado de los servicios (*enablers*) desplegados en Orion. Sin monitorización de los nodos de computación.	Desarrollo de una serie de *enablers* para habilitar una arquitectura IoT descentralizada.
aerOS	Desarrollo de un Meta-OS para la integración	Arquitectura descentralizada	Orquestación inteligente federada de	Si, el elemento Self-awareness desplegado	Federación de dominios y del sistema de orquestación de

	de los elementos heterogéneos del continuo.	*Cloud-Edge-IoT* basada en la federación de dominios que engloban diversos nodos de computación.	servicios en forma de contenedores. Dividida en 2 niveles: HLO y LLO.	obligatoriamente en cada nodo se encarga de monitorizarlo. Los datos de monitorización se envían al CB Orion-LD para ponerlos a disposición del Meta-OS de manera federada.	servicios.
FLUIDOS	Desarrollo de un Meta-OS utilizando herramientas establecidas en la industria para gestionar eficientemente los recursos que componen el continuo.	Arquitectura descentralizada basada en la gestión de diferentes clústeres de K8s y dispositivos IoT.	Orquestación de servicios basada en la reserva dinámica de recursos utilizando el protocolo propio REAR.	Monitorización de nodos de K8s utilizando Prometheus.	K8s *multi-cluster* mediante Liqo y gestión dinámica de recursos utilizando el protocolo REAR.
ICOS	Desarrollo de un Meta-OS distribuido diseñado para ser elástico, dinámico y móvil.	Arquitectura descentralizada *Cloud-Edge-IoT* en la que los nodos controladores gobiernan un grupo de nodos agente de manera horizontal, sin jerarquías claramente definidas.	Orquestación inteligente dividida en 2 niveles: controlador y agente.	Monitorización de nodos mediante la instalación del módulo *Logging and Telemetry*, que envía los datos a un módulo central que agrega de manera centralizada los datos enviados por los diferentes nodos del continuo.	Gestión descentralizada del continuo sin establecer relaciones jerárquicas a alto nivel entre nodos.

Capítulo 4

Diseño de la solución

4.1.Introducción

En los capítulos previos se ha realizado un amplio estudio de las tecnologías y tendencias existentes, entrando en más detalle en aquellas que pueden servir para construir la solución que se va a proponer en esta tesis doctoral. Asimismo, en el capítulo anterior se ha realizado un estudio similar, pero en términos más de implementación práctica, cimentado en un repaso cronológico de los proyectos de investigación en los que ha participado el doctorando, situándolos en el contexto tecnológico y de las áreas de investigación sobre los que se sustentan. Además, se enumera la experiencia personal adquirida durante el desarrollo de los mismos.

El trabajo realizado en la anterior sección habilitó al doctorando para poder llevar a cabo un proceso concienzudo de reflexión sobre las necesidades más latentes en el ámbito del continuo de computación *Cloud-Edge-IoT*, puesto que también se realizó un análisis de las iniciativas actuales de investigación y desarrollo promovidas por los diferentes grupos de acción de tanto la Comisión Europea como de entidades de referencia como la CNCF.

Es en este este punto en el que se puede afirmar que existe una brecha para alcanzar una implementación real, y con una base tecnológica suficientemente consolidada, del continuo de computación. De este estudio y su posterior reflexión se puede extraer la conclusión de que la herramienta tecnológica que mejor encaja para desarrollar un elemento uniforme de gestión sobre estos recursos de computación heterogéneos es un Meta-OS. El doctorando ha llegado a esta conclusión después de participar en el proyecto de investigación aerOS y descubrir todo el potencial que tienen este tipo de soluciones basadas en la construcción de un Meta-OS para el continuo. Además, esta afirmación está totalmente respaldada por la EC, puesto que no solo creó una iniciativa de proyectos de investigación con el objetivo de desarrollar este tipo de sistemas, identificando los Meta-OS como un elemento clave para habilitar el continuo *Cloud-Edge-IoT*, sino que apuesta por su continuidad en los próximos años [259][260].

A menudo, cuando se referencia tanto el continuo *Cloud-Edge-IoT* (y hasta de sus niveles *cloud, edge* o IoT por separado) como los sistemas para su gestión (como los Meta-OS), se tiende a identificar el principal desafío que se pretende resolver con el despliegue inteligente de servicios de manera distribuida a lo largo de los recursos de computación que conforman el continuo. Sin embargo, el desafío de implementar un mecanismo de gestión unificada de dicho continuo no solo se resuelve diseñando un sistema de despliegue inteligente de servicios de usuario (que hayan sido previamente desarrollados siguiendo patrones *cloud-native* o *edge-native*), sino que es necesario gestionar los nodos de computación reales que habilitan que el continuo sea posible de una manera uniforme, así como establecer patrones de conexión y colaboración entre ellos. Esta gran diversidad de nodos se refleja en la coexistencia de diferentes arquitecturas de CPU, sistemas operativos, herramientas de gestión y orquestación de contenedores, topologías y protocolos de red, entre muchos otros factores. Además, las localizaciones físicas de estos nodos son igualmente muy diversas, pudiendo estar distribuidas en diferentes regiones, países e incluso continentes, lo que añade un nivel adicional de complejidad al diseño y la gestión del continuo de computación CEI.

Esta gran diversidad de elementos coexistentes que no suelen ser considerados en una primera instancia, ya que se tiende a pensar en los elementos a más alto nivel como son las aplicaciones o servicios, se ilustra perfectamente utilizando un **iceberg a modo de símil**. Este símil, ilustrado en la Figura 4.1, permite visualizar cómo la verdadera complejidad del continuo reside en la capa subyacente, que incluye la interoperabilidad entre los diferentes tipos de elementos que entran en juego. Comprender y gestionar esta capa "invisible" es esencial para garantizar que los despliegues de aplicaciones sean efectivos y cumplan con los requisitos de los usuarios, sin importar las diferencias entre los nodos del continuo.

Después de esta reflexión, es necesario reconocer que actualmente no es posible homogeneizar todos los aspectos previamente descritos debido a numerosos factores, como pueden ser la presencia de soluciones ya desplegadas cuya migración no es factible, la existencia de una enorme cantidad de protocolos diferentes que siguen

estándares dispares o la falta de colaboración entre diversos equipos o grupos de trabajo, incluso pertenecientes a la misma empresa. La combinación de estos elementos genera un abanico de posibilidades prácticamente inabarcable. Por este motivo, la solución más lógica consiste en centrarse en los aspectos clave que permitan avanzar hacia la construcción de un Meta-OS funcional. Además, la elección de estos aspectos a resolver en última instancia dependerá del usuario final o del caso de uso que se pretenda resolver.

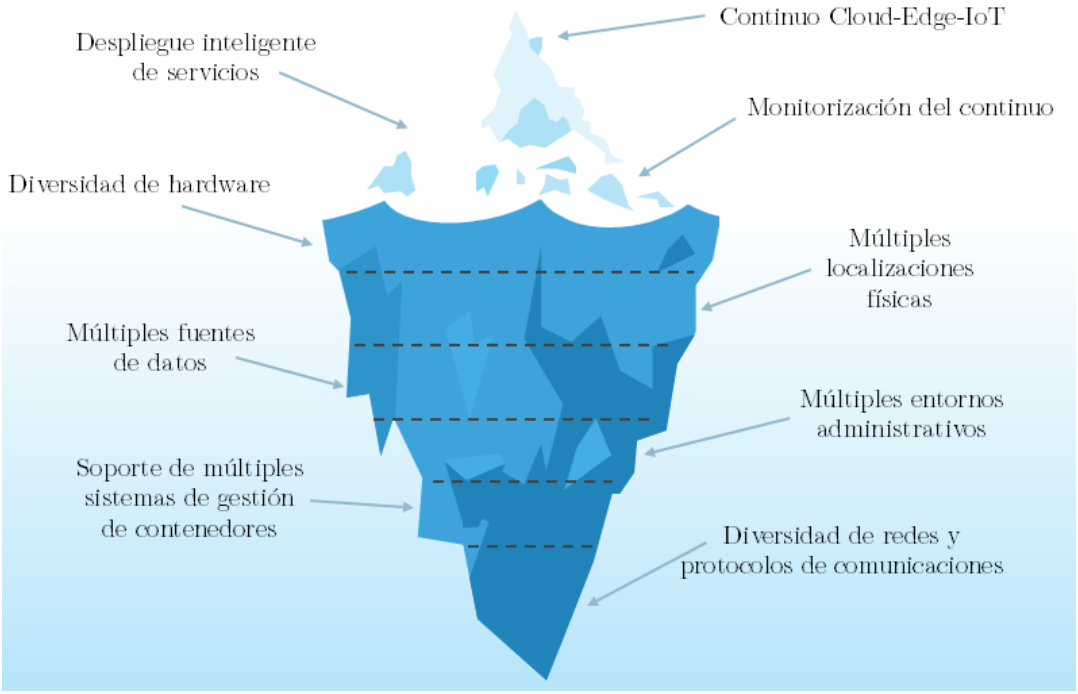


Figura 4.1: Símil del iceberg para ilustrar la complejidad del continuo CEI

En consecuencia, el diseño e implementación de un Meta-OS completo para gobernar el continuo de computación sería un objetivo demasiado ambicioso para ser realizado en una sola tesis doctoral, como se puede comprobar en los numerosos desafíos subyacentes que se ilustran en el anterior símil del iceberg. Esta afirmación también se basa en que la misma EC ha financiado mediante la inversión de decenas de millones de euros en diversos proyectos de investigación para este mismo fin, en los que participan numerosas empresas y estructuras de investigación consolidadas en el sector. Además, se pretende establecer una sinergia posterior para poner puntos en común sobre las soluciones desarrolladas y avanzar hacia un estándar unificado, afirmando de manera implícita que no hay una única solución válida y que las soluciones más ambiciosas o prometedoras deben salir de un consenso común y/o de un marco común de interoperabilidad (como tradicionalmente se ha abordado en el ámbito del IoT).

Por ende, el objetivo real de esta tesis doctoral se focaliza en el diseño y la materialización de los **componentes claves que habilitan un Meta-OS**, que han sido diseñados cuidadosamente con unos requisitos lo más generalistas posibles para que puedan encajar en soluciones basadas en las líneas de diseño de

arquitectura definidas en la TF3 (arquitectura) de la iniciativa EUCloudEdgeIoT. Además, la solución propuesta se construye sobre una serie de principios ampliamente establecidos o emergentes en el sector que se han extraído como resultado del trabajo realizado en la investigación del estado del arte, como son el uso de la virtualización y empaquetado en contenedores como unidad mínima de ejecución, uso de patrones de diseño *cloud-native* y *edge-native* convenientemente, o dotar de una gran flexibilidad a las operaciones de conexiones despliegue y comunicaciones entre servicios. La visión general de estos componentes claves que habilitan un Meta-OS se va a abordar en una subsección dedicada exclusivamente a esta materia.

Finalmente, en este capítulo, como su propio nombre indica, se van a abordar los conceptos y principios fundamentales en los que se basa la solución propuesta en esta tesis doctoral, cuya implementación se describirá en detalle en el capítulo 5. Este no es el único contenido que presenta el actual capítulo, puesto que la definición de requisitos, tanto funcionales como técnicos, es un proceso necesario para desarrollar una solución coherente y con unos objetivos claramente acotados. A modo de conclusión de esta sección, se detallará la metodología empleada para su diseño e implementación, partiendo desde un ámbito más general que abarca la tesis doctoral en sí misma para acabar con la metodología en el desarrollo de *software* y su despliegue en el continuo.

4.2. Conceptos de diseño fundamentales: nodos y dominios

Una vez ha quedado claramente definido el ámbito de la solución que intenta implementar esta tesis doctoral como respuesta a los desafíos que se han ido planteando, es necesario definir conceptualmente dos de los pilares básicos de diseño sobre la que ésta se sustenta: los nodos de computación y su agrupamiento en dominios. De esta manera, incluso antes de entrar en detalle en los diferentes requisitos que debe cumplir la solución propuesta, se pretende dotar de un mayor contexto al lector de esta tesis doctoral.

En el continuo de computación *Cloud-Edge-IoT* el elemento más granular que permite la ejecución de servicios o aplicaciones, prevé de capacidades de red y es a su vez una fuente de datos, es un **nodo o elemento de computación** [261]. El establecimiento de esta unidad mínima es de suma importancia para construir sobre ella el proceso y las funcionalidades de orquestación de servicios y red, así como de monitorización. Estos nodos del continuo son evidentemente heterogéneos, al igual que el propio continuo de computación, puesto que son sus unidades básicas, pero al mismo tiempo comparten varias características [261]. Por ende, las propiedades que debe cumplir un recurso de computación para ser considerado como un nodo de pleno derecho del continuo son las siguientes:

- Capacidad para ejecutar servicios virtualizados en forma de contenedores.

- Existencia de un OS subyacente en el que se pueda instalar *software*, así como disponer de cierto nivel de control sobre él.
- Disponibilidad de capacidades de computación, almacenamiento y de red.
- Disponibilidad de interfaces de red que puedan ser controladas para establecer conexiones otros nodos, a nivel de continuo o al menos de dominio.
- Capacidad de exponer ciertos servicios desplegados en él para que sean accesibles por los demás actores del continuo.

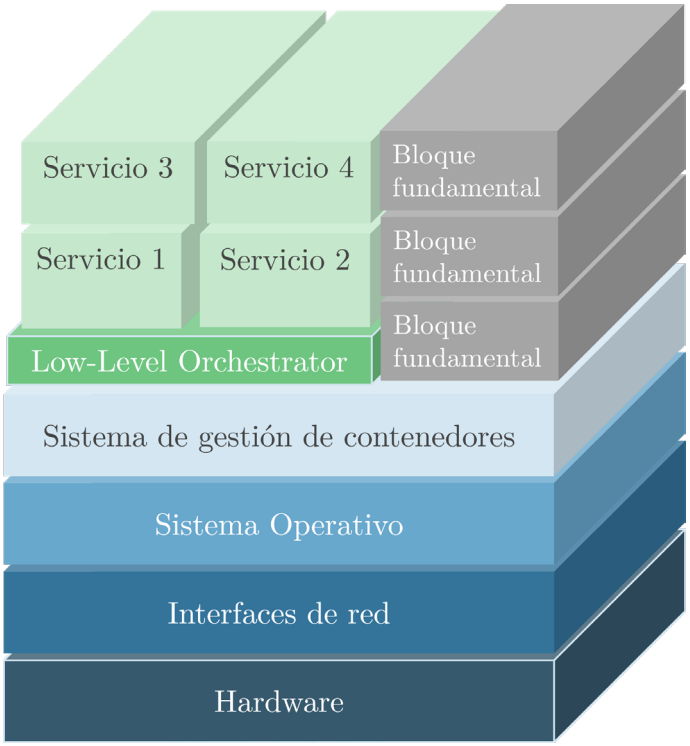


Figura 4.2: Nodo del continuo de computación

Antes de continuar con la descripción de un nodo, conviene detenerse brevemente en el que puede considerarse el primer requisito fundamental para que un recurso de computación sea considerado como un nodo autónomo dentro del continuo: la capacidad de ejecutar contenedores. Este requerimiento se ha establecido después del análisis realizado durante la investigación del estado del arte, a partir de la cual se puede concluir que el uso de contenedores como técnica de virtualización se ha convertido en un estándar *cloud-native*, y que se han desarrollado las suficientes herramientas para que pueda ser trasladado a los equipos con recursos propios del *edge computing*. Además, el uso de contenedores abre el abanico de las posibles herramientas punteras que se pueden utilizar para intentar homogeneizar el continuo de computación. Por lo tanto, la unidad mínima de empaquetamiento o virtualización para desplegar tanto los componentes claves que habilitarán el Meta-OS desarrollados durante esta tesis como los servicios de los usuarios que se pretendan desplegar en el continuo, se establece en contenedores.

Una vez se ha establecido esta unidad mínima de despliegue, es necesario realizar una aclaración importante. Como también se ha descrito a lo largo de la sección 2.6, los contenedores están realmente controlados en última instancia por un *container runtime*, pero a nivel práctico es más usual el uso de *container engines* (como Docker o Balena) o sistemas de orquestación que actúan sobre estos *container runtimes* (como K8s), siendo extraño el caso en el que un usuario final interactúa directamente con estos *runtimes*. En consecuencia, se ha decido usar el término **"sistema de gestión de contenedores"** a lo largo de esta tesis con el propósito de unificar todas las herramientas de gestión de contenedores (Docker, containerd, K8s, Balena...), pero sin olvidar en ningún momento las diferencias subyacentes que existen entre todas estas tecnologías.

Volviendo a la definición de un nodo del continuo, cualquier tipo de recurso de computación o equipamiento *hardware*, ya sea físico o virtualizado, que cumpla con las anteriores características, puede ser considerado como un nodo "de pleno derecho" del continuo de computación que se concibe en esta tesis doctoral. Se puede afirmar sin lugar a duda que una gran diversidad de equipamiento cumple con las propiedades previamente descritas, siendo claros ejemplos ilustrativos los siguientes: (i) nodos que conforman un clúster de K8s (tanto nodos *master* como *workers*, dependiendo de la configuración del clúster), (ii) máquinas virtuales con un *container runtime* (Docker, containerd...) instalado, (iii) dispositivos *edge* como SBCs (p. ej. Raspberry Pi o la GWEN de ASSIST-IoT) con un OS de propósito general (p. ej. Raspberry Pi OS o Ubuntu) y un *container runtime* instalados o (iv) con un OS enfocado a la ejecución de contenedores (p. ej. Balena o Pantacor), y finalmente v) nodos *edge* pertenecientes a un sistema de orquestación alternativos a K8s como KubeEdge. Como se ha podido observar, en el caso de K8s, se ha decidido que el elemento mínimo de computación sea un nodo en lugar de considerar todo el clúster como un solo nodo del continuo para alcanzar un mayor nivel de granularidad.

Los ejemplos anteriores permiten entender que el concepto de nodo o elemento de computación se ha establecido para denominar recursos que pueden existir a lo largo de todo el continuo, abarcando desde dispositivos con recursos limitados hasta centros de datos locales y grandes equipamientos de la nube, ya cuenten con características más tradicionales o de nueva generación. De esta forma, se sienta la base para una gestión uniforme de la complejidad subyacente del continuo, facilitando su orquestación eficiente y coherente.

Respecto a este equipamiento que conforma el continuo, es necesario considerar un caso específico y relevante: los dispositivos IoT. Estos dispositivos son una pieza clave del continuo, puesto que no solo proporcionan métricas clave (mediciones de sensores, datos de monitorización y de contexto, etc.), sino que al mismo tiempo que ejercen como elementos activos (reciben órdenes para ejecutar ciertas acciones o directamente pueden ejecutar servicios demandados por los usuarios). En este caso, si los dispositivos IoT cuentan con una capacidad suficiente para cumplir con los requisitos descritos previamente (ejecutar contenedores, contar

con un OS configurable...), han de formar parte del continuo como un elemento de computación más. Por el contrario, si no cumplen con alguno de dichos requerimientos, los dispositivos IoT deben ser asignados a un nodo para establecer una relación de sinergia con él. Este es el caso habitual para dispositivos como sensores, actuadores, máquinas industriales o simples *gateways* IoT, que se comunicarían con el nodo más próximo (por ejemplo, un equipamiento como capacidad de E/S para conectar sensores directamente o usando algún protocolo específico, como un receptor ZigBee) para enviar o recibir datos e integrarse en el continuo.

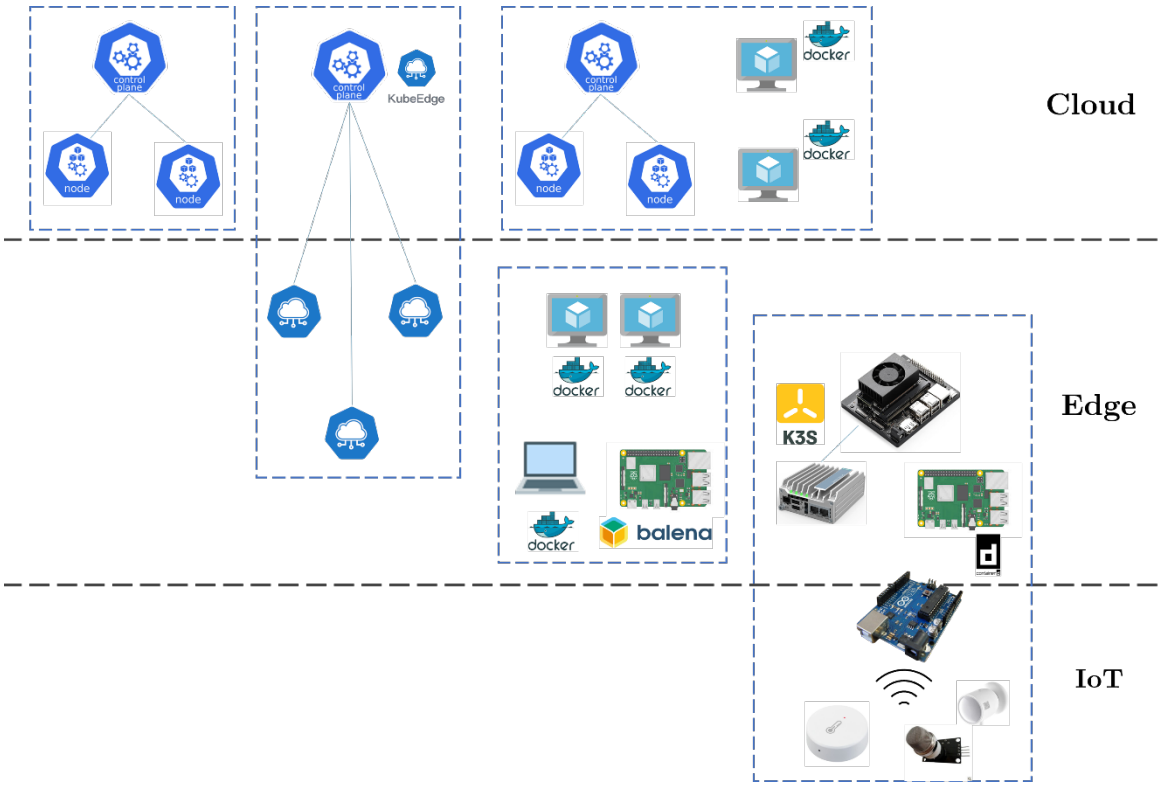


Figura 4.3: Ejemplo de un continuo *Cloud-Edge-IoT* formado por diferentes dominios

Después de definir el concepto de nodo o elemento de computación dentro de la visión de continuo de computación de esta tesis doctoral, es necesario explorar el agrupamiento de estos nodos en diferentes elementos conceptuales llamados **dominios**. Este agrupamiento, cuya identificación topológica debe realizarse como paso previo a la instalación del Meta-OS, es necesario para tratar de organizar los múltiples nodos que conforman el continuo en entidades administrativas de acuerdo con ciertos criterios lógicos y técnicos, que además compartirán la misma instancia de los bloques o funcionalidades esenciales del Meta-OS que se pretende desarrollar. No obstante, queda patente la dificultad de realizar este agrupamiento de manera uniforme y con criterios administrativos y objetivos. En consecuencia, el doctorando opina que la mejor opción consiste en dar una serie de orientaciones, pero siempre dejando en último lugar la decisión en manos de los administradores o propietarios de la infraestructura para que cuenten con una total libertad de diseño del continuo,

puesto que estos con total seguridad disponen de un mayor nivel de conocimiento sobre los elementos subyacentes del mismo (conexiones de red, capacidades de computación, desempeño histórico, problemas frecuentes, etc.). En la Figura 4.3 se muestra un ejemplo de continuo con sus dominios delimitados.

Como referencia, la decisión de la creación de dominios puede estar guiada por los siguientes criterios:

- **Geográficos:** una solución bastante lógica y directa consiste en agrupar todos los nodos que pertenecen a un área geográfica específica en un mismo dominio. Esta división es típica en entornos *cloud* donde los proveedores de servicios especifican ciertas áreas geográficas en las que están situados los servidores o centros de datos. Otro ejemplo sería en el caso de una universidad como la UPV que cuenta con diferentes campus, de manera que los nodos pertenecientes a cada uno de los campus formarían un dominio independiente.
- **Administrativos:** se trata del criterio de división más recomendable cuando los elementos de computación pertenecen a diferentes organizaciones o subdivisiones de la misma entidad, como el anterior ejemplo de la UPV que cuenta con diferentes campus, escuelas y departamentos. Aunque en última instancia se compartan los recursos de computación gestionados por el Meta-OS, algunos aspectos operacionales o de gestión más concretos podrían permanecer independientes.
- **Sistemas de gestión de contenedores:** se trata de un criterio más tecnológico, en el que se tienen en cuenta el sistema de gestión de contenedores utilizado por los nodos. Es importante señalar que este criterio no significa que en un dominio solo pueda haber nodos con un mismo tipo de sistema de gestión de contenedores, ya que más bien es todo lo contrario. Por ejemplo, se podría tratar un clúster de K8s como un único dominio por las facilidades de configuración que aporta esta tecnología, al igual que un clúster de KubeEdge, o combinar ambos con varios nodos que cuenten con solamente un *container runtime*.
- **Criterios de conectividad y de estructura de la red:** este criterio sería interesante para entornos en los que se necesita conectar servicios distribuidos entre nodos que compartan una misma red.
- **Capa del continuo:** consiste en dividir los dominios de acorde a la capa del continuo en el que se encuentren los nodos de manera jerárquica. En este caso se tendrían dominios claramente *cloud, edge* o *IoT*.
- **Operacionales:** esta división depende claramente del caso de uso. Por ejemplo, en una línea de producción industrial, un dominio podría comprender todos los elementos que pertenecen a la misma línea de producción o área de trabajo.

- **Otros:** se podrían agrupar nodos por su tipo de *hardware* (SBCs, equipamiento industrial, servidores...), entre muchos otros criterios posibles.

En casos de uso reales, es esperable que se combinen varios de estos criterios a la hora de diseñar los dominios del continuo de computación en el que se desplegaría el Meta-OS desarrollado, tal y como se podrá comprobar en los diferentes dominios que se han diseñado en los casos de uso expuestos en la sección 6.4 de esta tesis. Además, esta libertad de diseño permite que esta acción previa al despliegue del Meta-OS no suponga una gran limitación o una importante barrera de entrada para la adopción del mismo.

4.3. Análisis de requisitos

Como se ha indicado en las secciones anteriores, el principal objetivo de esta tesis es el diseño y el desarrollo de los componentes claves que habilitan un Meta-OS. Este objetivo se enmarca en un contexto donde el desarrollo de *software* tiene un papel fundamental, por lo que siguiendo las buenas prácticas en el desarrollo TIC, la captura de requisitos previamente a comenzar el desarrollo es esencial [262]. Este proceso se ha hecho de acuerdo con la metodología Volere [263] debido a que ha sido utilizada en los proyectos de investigación previamente mencionados. En resumen, Volere es una metodología para la captura y gestión de requisitos, basada en una plantilla estructurada que abarca aspectos funcionales, no funcionales y de gestión, garantizando trazabilidad y calidad en el desarrollo de sistemas.

En esta sección, se describen los requisitos técnicos y funcionales que deben cumplir los diferentes bloques del Meta-OS para garantizar su correcto funcionamiento, interoperabilidad y capacidad de adaptación a un entorno altamente distribuido y heterogéneo. Estos requisitos establecen las bases para asegurar que cada componente no solo cumpla con su propósito específico, sino que también se integre de manera directa con la solución general, contribuyendo al cumplimiento de los objetivos generales del sistema.

En resumen, se van a describir estos dos tipos de requisitos:

- **Requisitos técnicos:** este tipo de requisitos son de un ámbito más específico o de más bajo nivel, ya que ponen las bases del diseño, desarrollo e implementación de los diferentes componentes que constituyen el sistema desarrollado. Por ejemplo, incluyen el uso de una cierta tecnología (p. ej. contenedores) o la definición de dependencias o interacciones necesarias entre componente.
- **Requisitos funcionales:** estos requisitos definen las funcionalidades y servicios que el sistema desarrollado debe proporcionar para satisfacer los objetivos de la tesis. En resumen, tienen un perfil de más alto nivel o general, poniendo el foco en el comportamiento observable del sistema.

4.3.1. Requisitos técnicos

En la siguiente tabla se definen los requisitos técnicos capturados junto con una descripción detallada de los mismos.

Tabla 4.1: Requisitos técnicos de la solución

Requisito	Descripción
Arquitectura descentralizada basada en la federación de dominios	El continuo de computación que se concibe en esta tesis debe tener una arquitectura descentralizada y basada en la federación de dominios (formados a su vez por un conjunto de nodos de computación). Esta federación se debe materializar en varios niveles del Meta-OS: información de contexto clave, orquestación de servicios, comunicaciones e intercambio de datos.
Uso de contenedores como unidad mínima de despliegue	La unidad mínima de empaquetamiento o virtualización para desplegar tanto los componentes desarrollados durante esta tesis como los servicios de los usuarios que se pretendan desplegar en el continuo, se establece en contenedores.
Compatibilidad con diversos sistemas de gestión de contenedores	Debido a que la unidad mínima de despliegue se ha establecido en contenedores, el Meta-OS debe ser compatible con diversos sistemas de gestión de contenedores, sobre todo para el despliegue final de servicios de usuario en los nodos seleccionados por el proceso de orquestación. Además, esta compatibilidad debe ser fácilmente ampliable a nuevos sistemas. Respecto a la instalación del Meta-OS, solo se contemplan dos de los sistemas más establecidos: Docker y Kubernetes.
Instalable sobre infraestructura existente	El punto de partida de la instalación del Meta-OS debe ser infraestructura debidamente preparada para cumplir con los requisitos que se han establecido para ser considerada como un nodo del continuo (sistema de gestión de contenedores, OS, conexión a una red...). A diferencia de otras soluciones, el ámbito de aplicación de esta solución no está diseñada para operar sobre una infraestructura sin una configuración previa. De esta manera, se pretende mejorar la adopción de la solución.
Escalabilidad y flexibilidad	El sistema desarrollado debe habilitar un continuo de computación que sea altamente escalable y flexible. Un claro ejemplo es la creación, edición y eliminación de

	dominios permitiendo un moldeamiento sin complicaciones del mismo.
Definición y uso de modelos de datos comunes y estandarizados	Es necesario que la solución haga uso de modelos de datos comunes a todos sus componentes, ya sea de nueva creación (modelo de datos del continuo) o especificaciones previamente estandarizadas (CloudEvents de la CNCF o Smart Data Models de FIWARE). Este requerimiento aplica tanto para el intercambio de datos como en las visualizaciones mostradas a los usuarios finales.
Observabilidad en varios niveles	Los procesos claves que habilitan las funcionalidades básicas del Meta-OS deben ser completamente transparentes y trazables, aportando observabilidad en varios niveles.
Fidelidad con la iniciativa de estandarización de EUCEI	Los procesos claves del Meta-OS han de seguir las recomendaciones y arquitectura establecidas en los WGs de la iniciativa EUCEI con la mayor fidelidad posible. Un caso concreto es el proceso de orquestación inteligente de servicios, que debe estar dividido en dos niveles y habilitado por la federación.
Logging estructurado en los componentes desarrollados	A diferencia del *logging* tradicional, que utiliza simples mensajes de texto, el *logging* estructurado organiza la información en un formato que facilita su análisis automatizado y búsqueda eficiente. Por ende, los componentes desarrollados deben hacer uso de librerías de *logging* estructurado que creen mensajes en formato JSON.
Estrategia de empaquetado común	Los componentes desarrollados han de seguir una estrategia de empaquetado común, ya sea en Helm charts o en archivos Docker Compose, cuyo proceso puede ser automatizado. Esta estrategia incluye el uso de etiquetas de identificación comunes.
Patrones de diseño *cloud-native* y *edge-native*	Los componentes desarrollados deben seguir los patrones de diseño *cloud-native* o *edge-native* convenientes dependiendo de su ámbito de aplicación.
Uso de herramientas de código abierto ampliamente establecidas	Los componentes desarrollados han de hacer uso de librerías, tecnologías o protocolos de código abierto ampliamente establecidos en la industria y soportados por entidades de referencia como la CNCF.

4.3.2. Requisitos funcionales

En la siguiente tabla se definen los requisitos funcionales con los que debe contar el sistema, junto con una descripción detallada de los mismos.

Tabla 4.2: Requisitos funcionales de la solución

Requisito	Descripción
Flexibilidad para el diseño del continuo	El diseño de los dominios que conforman el continuo (nodos que forman parte de él, elección de un nodo principal o externo, selección del dominio *entrypoint*...) debe ser altamente flexible, de manera que los administradores o propietarios de la infraestructura cuenten con un alto grado de libertad. El único requisito estricto es que al menos un dominio tiene que contar con una IP pública o que sea accesible por todos los demás dominios. A modo de ayuda para la realización de este cometido, se proporcionan una serie de orientaciones.
Abstracción de la complejidad subyacente del continuo	El Meta-OS debe intentar abstraer en la medida de lo posible la complejidad subyacente del continuo de computación, es decir, la heterogeneidad de nodos que conforman los dominios o la diversidad de sistemas de gestión de contenedores que presentan.
Plataforma unificada de gestión del continuo	El Meta-OS debe contar con una plataforma única para gestionar el continuo de computación de una manera unificada, siempre teniendo en cuenta la naturaleza descentralizada del continuo. Esta plataforma debe actuar como el punto de entrada único de gestión y observabilidad del continuo, recabando información y ejerciendo acciones de manera federada.
Monitorización continua de los recursos de computación	Los recursos de computación proporcionados por los nodos que forman el continuo debe ser monitorizada continuamente y en tiempo real para dotar a los administradores de una visión general y completa del continuo. Asimismo, esta información se debe mostrar en una interfaz gráfica amigable que ayude a identificar las métricas clave.
Ubicuidad en el acceso a la información clave del continuo	La información clave del continuo CEI (como las métricas de monitorización de los recursos de computación) debe ser accesible de manera ubicua, es decir, desde cualquier punto del continuo con independencia del dominio o localización, fortaleciendo la naturaleza descentralizada del Meta-OS.

Despliegue inteligente de servicios	El sistema debe incorporar un sistema de orquestación de servicios, soportado por modelos inteligentes, para desplegar los servicios solicitados por los usuarios en los nodos más indicados teniendo en cuenta los requisitos indicados por ellos.
Especificación de requisitos en la orquestación de servicios	El usuario debe poder introducir los requisitos de funcionamiento (consumo de recursos, necesidades específicas...) y de despliegue de los servicios (condiciones de exposición de red...) que pretenda desplegar en el continuo de una manera clara y sencilla.
LCM completo de los servicios desplegados	El Meta-OS debe contar con un sistema para la gestión del ciclo completo de vida (LCM – *Lifecycle management*) de todos servicios desplegados por el orquestador del Meta-OS. Este requisito implica la monitorización continua de su estado en varios niveles (tanto a nivel global o de Meta-OS como a nivel interno mediante el sistema de gestión de contendores que lo ha desplegado), incluyendo el reporte de errores de ejecución.
Sistema de notificaciones y alarmas	La solución debe contar necesariamente con un sistema de notificaciones y alarmas en tiempo real para poner en conocimiento a los usuarios de ciertos eventos claves que se den en el continuo (p. ej. la adición o eliminación de un nodo o dominio, el despliegue de un servicio o el fallo de un componente). Además, estas notificaciones deben constar en un registro que pueda ser consultado por los usuarios.

4.4. Metodología

En este apartado se pretende realizar una explicación de la metodología general que se ha seguido para desarrollar la solución propuesta durante esta tesis doctoral, es decir, los componentes claves que habilitan un Meta-OS para gestionar la complejidad del continuo de computación. A modo de resumen, se ha realizado un esquema (Figura 4.4) en el que se pueden visualizar claramente la relación entre los diferentes procesos, representados en forma de bloques.

El punto de partida de esta tesis doctoral es sin lugar a duda la concreción del plan de investigación. La realización de este trabajo previo es sumamente importante, puesto que sienta las bases del proceso de investigación, desarrollo e implementación de la tesis, además de ayudar a acotar su ámbito de aplicación, permitiendo elaborar una serie de líneas maestras sobre las que construir la solución finalmente implementada. Este plan de investigación estuvo en gran parte

influenciado por la experiencia previa en proyectos de investigación en los que el doctorando ha podido comprobar de primera mano los principales desafíos que se han listado en el apartado de motivaciones y objetivos de esta tesis.

La ejecución del plan de investigación empezó con el análisis del estado del arte para construir una base sólida sobre la que sustentar la tesis e intentar aportar resultados originales. En esta investigación también influyó en gran manera la participación en proyectos de investigación, puesto que el doctorando no solamente ha podido constatar los conceptos descritos y hacer uso de algunas de las herramientas, sino que muchas de ellas han servido como punto de partida para descubrir tecnologías nuevas o para abrir nuevas líneas de investigación. Otro aspecto importante a considerar es la escritura de artículos de investigación y la participación en congresos y conferencias en los que el doctorando tuvo la oportunidad de exponer los resultados de su investigación, proceso que a su vez permitió establecer una relación de sinergia con otros trabajos gracias a la citación de los mismos y al intercambio de ideas en dichos congresos. Además, esta fase también permitió acotar la ambición de esta tesis hasta dar con un objetivo final claro y relevante (diseño y materialización de los componentes claves que habilitan un Meta-OS).

Después de realizar este proceso más teórico o de investigación pura, se realizó el diseño completo de la solución. Este diseño ha estado basado en todo momento en las líneas maestras y en los conceptos adquiridos durante dicho proceso, siendo totalmente fiel a estándares como el propuesto por la iniciativa EUCEI para los sistemas de orquestación inteligente de servicios en el continuo, en la que el doctorando ha tenido una participación clave.

La siguiente acción consistió en el desarrollo e implementación real de la solución antes diseñada, que evidentemente implica un proceso de desarrollo de *software* para implementar los componentes necesarios para habilitar el Meta-OS. Esta fase de desarrollo está también influenciada por los requisitos técnicos y funcionales descritos en el apartado anterior. Además, se estableció como imperativo el uso de las tecnologías y herramientas de código abierto estudiadas durante el estado del arte, como los proyectos bajo el paraguas de la CNCF (NATS, K8s, Helm...). La metodología específica seguida durante el proceso del desarrollo del *software* se describe en una subsección específica dada importancia de la materia.

Cuando la fase de implementación alcanzó un nivel de madurez suficiente, comenzó a compaginarse con actividades de integración y testeo, en gran parte realizadas sobre la infraestructura del grupo SATRD, descrita en el punto 6.2. Finalmente, cuando finalizó la etapa de desarrollo, el Meta-OS resultante se desplegó (siempre en su versión más actual) en diferentes casos de uso para validarlo, al mismo tiempo que mostrar la versatilidad del sistema desarrollado durante la tesis. Esta metodología también permitió detectar fallos de una manera temprana al establecer una exitosa relación de retroalimentación, además de mejorar la usabilidad de ciertas funcionalidades a nivel de usuario final.

Para concluir con la metodología, el estudio del estado del arte aportó nuevas ideas para la extensión de la solución con la inclusión de técnicas alternativas de virtualización que, o bien no están lo suficientemente adoptadas, o a pesar de ser prometedoras aún no han alcanzado un grado de madurez suficiente para ser contempladas en escenarios de validación reales. Por lo tanto, de manera adicional se realizó un trabajo de prospectiva sobre la solución final para explorar el encaje de estas nuevas tecnologías y la posibilidad de abrir nuevas líneas de investigación.

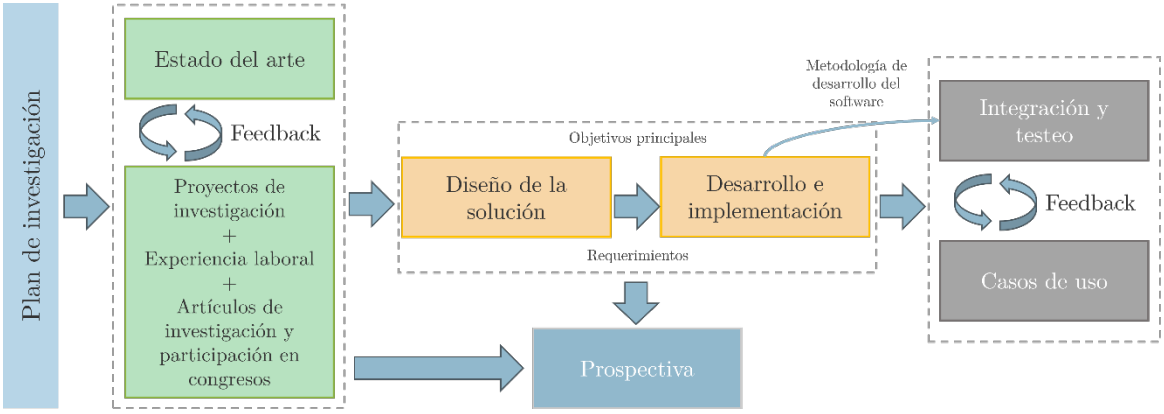


Figura 4.4: Metodología de desarrollo de la solución propuesta

4.4.1. Desarrollo de *software*

La mayoría de los desarrollos de *software* empieza en la máquina personal del desarrollador o del grupo de desarrolladores, y a medida que va alcanzando un cierto nivel de madurez, se va publicando en repositorios de código, utilizando herramientas de gestión de código (siendo *git* la más habitual), para mantener un seguimiento de su progreso, facilitar su desarrollo de manera colaborativa o simplemente para implementar un sistema de respaldo. En el caso de esta tesis doctoral, se ha hecho uso del repositorio GitLab privado del grupo de investigación SATRD para no depender de repositorios públicos como GitHub durante la fase de desarrollo privada o interna. Esta decisión también se justifica en que, aunque GitHub ofrece en su plan personal y gratuito un número ilimitado de repositorios privados, existe una limitación en cuanto al tiempo de ejecución de los procesos enmarcados dentro de las *pipelines* de CI/CD (*Continuous Delivery and Continuous Deployment*). Esta limitación se ha identificado como un potencial riesgo que se ha preferido evitar de manera preventiva, dado que podría haber interferido con los tiempos de desarrollo e integración de los componentes *software* desarrollados durante la tesis.

GitLab es una herramienta con la suficiente adopción en la industria, y además proporciona una versión de código abierto bajo la licencia MIT, sin incluir ninguna diferencia de base con su versión de pago, excepto algunas opciones de configuración adicionales que no se podrían calificar como imprescindibles [264].

Los repositorios de código de GitLab no solo incluyen las funcionalidades esperables vinculadas al código fuente en sí mismo (creación de ramas, *releases*, *merge* o *pull requests*...), sino que también incluyen un registro para almacenar imágenes de contenedores (*container registry*) y un gestor de paquetes para almacenar Helm charts o ficheros Docker Compose (*package registry*), entre otras funcionalidades. Este factor diferencial es importante para el proceso de desarrollo del sistema propuesto en la tesis, puesto que como se ha indicado en las secciones previas, su unidad mínima de despliegue son contenedores. Por ende, las imágenes de contenedores creadas a partir del código fuente, así como los Helm charts en los que se han empaquetado los diversos componentes para ser desplegados en los clústeres de K8s que formaran parte del continuo, pueden almacenarse en el mismo repositorio de código que se ha utilizado para su desarrollo. Esta doble naturaleza de los repositorios permite evitar el uso de registros de imágenes de contenedores o repositorios de Helm charts adicionales, ya sea públicos (Docker Hub o Artifact Hub) o privados (Harbor).

Además, este proceso se puede automatizar haciendo uso de metodologías DevOps (como la metodología DevPrivSecOps propia desarrollada por el proyecto aerOS [265]), mediante las cuales se pueden crear automáticamente las imágenes de contenedores y los Helm charts asociados a un desarrollo, e incluso se pueden incluir técnicas de GitOps [266] para que estos charts o contenedores se desplieguen automáticamente en clústeres de K8s, y de esta manera mantener actualizados los componentes del Meta-OS de manera automática.

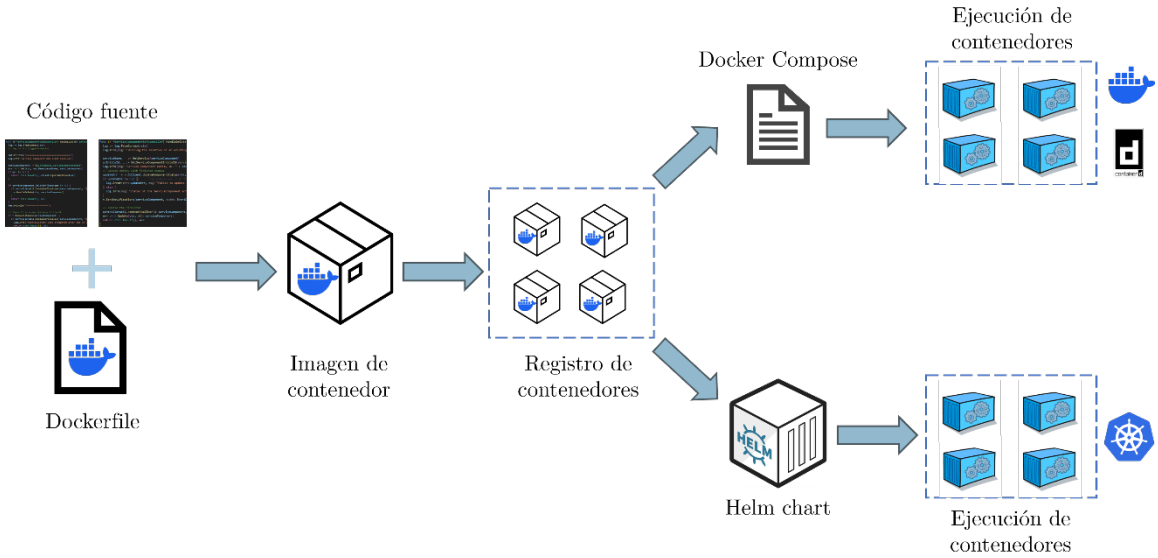


Figura 4.5: Etapas de desarrollo del *software* hasta su ejecución en el continuo

Después de seleccionar GitLab como repositorio de desarrollo, el doctorando valoró que era necesario desarrollar una estrategia de empaquetado y despliegue del *software* desarrollado para que su proceso de testeo fuera más eficiente y de esta manera acortar el tiempo invertido en tareas repetitivas. Aunque cada repositorio de GitLab cuenta con sus propios registros, la elección ideal no consiste en usar los

registros internos de cada repositorio para instalar un único componente, más aún cuando una estrategia común de desarrollo en el ámbito de sistemas distribuidos (que frecuentemente están compuestos por varios microservicios) es utilizar un solo repositorio para cada microservicio o módulo del componente. Por lo tanto, se decidió crear un repositorio común para almacenar todo el *software* empaquetado previamente. Entrando en detalles, el propósito de dicho repositorio no fue el almacenamiento de código fuente (de hecho, no contiene ningún archivo de código), sino de componentes ya empaquetados como imágenes, archivos Docker Compose, manifiestos de K8s o Helm charts, en su propio *package* y *container registry*. Este repositorio común se ha utilizado para desplegar el sistema desarrollado en los casos de uso descritos durante el capítulo 6.

Finalmente, es necesario recordar que, como se ha apuntado desde el principio de esta memoria, el doctorando mantiene un fuerte compromiso con la utilización, fomento y producción de código y tecnología *open source*. Por este motivo, existe el compromiso de publicar todo el código desarrollado durante la realización de esta tesis en repositorios públicos de GitHub bajo las licencias de código abierto (Apache 2.0, GPLv3, MIT...) más adecuadas para cada desarrollo, permitiendo su inspección por parte de la comunidad y así ser objeto de posibles mejoras, adaptaciones o reporte de errores. Además, en línea con el contenido anterior, para permitir su utilización también se pretende publicar directamente los elementos empaquetados previamente mencionados (imágenes, *charts*...). La elección obvia para la publicación de las imágenes de contenedores es Docker Hub (el registro de contenedores por defecto de K8s, como se ha apuntado en la sección 2.6.7), mientras que Artifact Hub uno de los mejores repositorios para publicar Helm charts. No obstante, aunque existen varias iniciativas como el repositorio *awesome compose* en GitHub [267], actualmente no existe un repositorio destacable con el fin de la publicación exclusiva de ficheros de despliegue Docker Compose. Por lo tanto, se pretende explorar su publicación en un repositorio de GitHub dedicado únicamente a este fin, incluyendo estos ficheros tanto en el repositorio de código como en una GitHub page asociada.

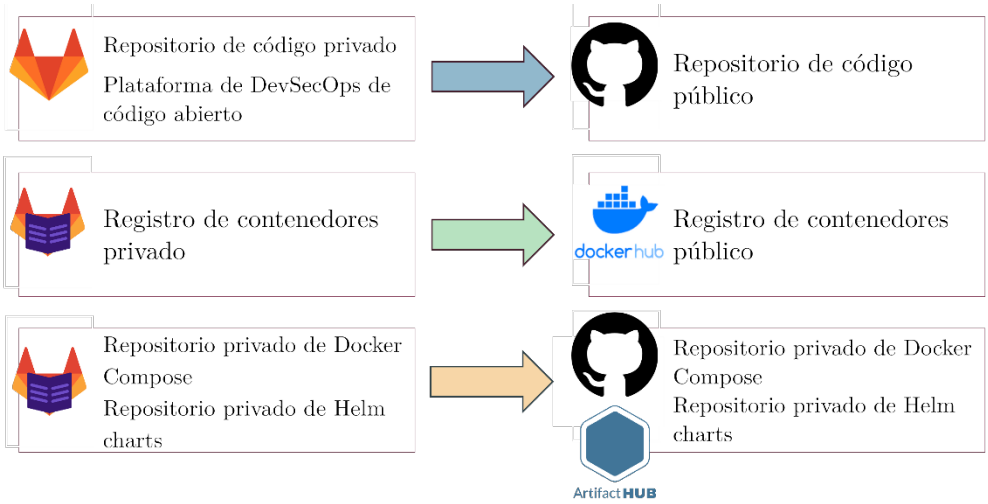


Figura 4.6: Etapas de publicación del *software* desarrollado

4.5.Bloques esenciales del Meta-OS

En la introducción a este capítulo, se ha justificado el establecimiento del objetivo de esta tesis en el diseño y la materialización de los componentes claves que habilitan un Meta-OS. Por lo tanto, para concluir esta sección se van a escribir de manera pormenorizada cada uno de estos bloques fundamentales, de manera que también sirva a modo de enlace con el siguiente capítulo, en el que se describirá el desarrollo de los componentes que conforman a los mismos con todo tipo de detalles técnicos. En la siguiente figura se puede observar la estructura de estos bloques de una manera esquemática y visual, como introducción al contenido que se va a abordar.

Por último, es necesario recordar que este Meta-OS se despliega y se construye sobre una serie de infraestructura existente que ha sido previamente instalada y configurada.

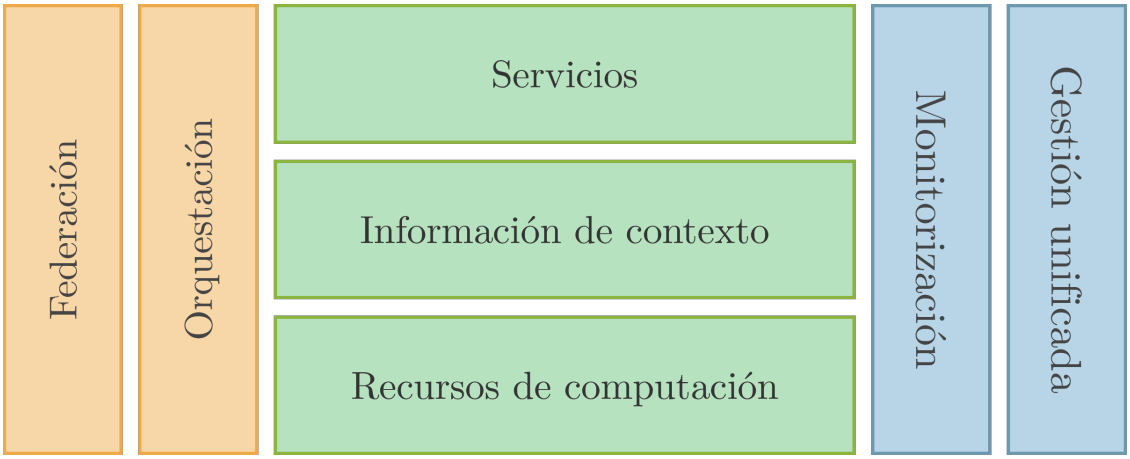


Figura 4.7: Bloques esenciales que habilitan un Meta-OS para gobernar el continuo

4.5.1. Federación

Los sistemas con una arquitectura federada representan un cambio de paradigma en el ámbito de los sistemas de computación, al proporcionar un marco de interacción más igualitario, en el que todos los nodos dentro de un mismo sistema operan de manera colaborativa como iguales, contando con el mismo nivel de acceso a los recursos y con capacidades similares en cuanto a la toma de decisiones. Este enfoque no solo fortalece la autonomía, sino que también permite un control más granular sobre los recursos distribuidos, transformando de manera fundamental el ecosistema de orquestación y gestión de recursos.

En el entorno altamente heterogéneo del continuo *Cloud-Edge-IoT*, los mecanismos de federación deben operar sobre entidades de alto nivel que agrupen múltiples nodos subyacentes, que en este caso son los dominios anteriormente descritos. De no ser así, la federación resultaría inviable, ya que implicaría gestionar

una cantidad excesiva de entidades o nodos individuales, especialmente en las capas *edge* y *far-edge*, donde el número de dispositivos involucrados es considerable.

Si bien este enfoque de federación a nivel de dominio introduce un cierto grado de centralización local en lo que respecta a los nodos pertenecientes a un mismo dominio, puede afirmarse que este esfuerzo es esencial para avanzar hacia una arquitectura más descentralizada y colaborativa en el contexto global del continuo de computación. En resumen, se trata de realizar un ejercicio de abstracción, centralización o agrupamiento en primer lugar, que puede ser visto como el precio a pagar para poder avanzar hacia una mayor descentralización del sistema a nivel global.

Lamentablemente, la federación de dominios también conlleva desafíos adicionales. En primer lugar, un sistema más distribuido implica un incremento en la latencia de las comunicaciones, ya que los datos clave deben compartirse de manera cooperativa entre todos los actores. Por lo tanto, este obstáculo debe abordarse de manera eficiente para evitar posibles cuellos de botella a medida que el sistema vaya escalando. En segundo lugar, este intercambio de datos requiere el establecimiento de comunicaciones directas entre todos los dominios, lo que implica que cada dominio debe ser accesible desde los demás mediante el establecimiento de una infraestructura de red compartida. Por último, el tráfico de red es un factor altamente sensible, puesto que puede incrementarse como consecuencia del aumento de las comunicaciones necesarias entre estos dominios.

A juicio del doctorando, el bloque de federación es uno de los pilares fundamentales sobre el que se sustenta la innovación que pretende aportar esta tesis doctoral. Esta afirmación se fundamenta en que este bloque supone un primer paso para el establecimiento de comunicaciones y relaciones entre entidades iguales en ámbitos fundamentalmente distribuidos, sin que los elementos subyacentes que realmente lo componen respondan a patrones similares en cualquiera de los niveles.

El bloque de federación del Meta-OS que se propone en esta tesis doctoral actúa principalmente sobre tres elementos del continuo de computación:

- **Servicios**: los servicios requeridos por los usuarios deben poder desplegarse en cualquier nodo de computación sin importar el dominio del que formen parte. Por ende, el Meta-OS debe integrarlos en el continuo sin establecer diferencia alguna entre ellos, permitiendo su interconexión y el intercambio de datos entre ellos.
- **Información de contexto**: la información de contexto global del continuo debe ser accesible con independencia del dominio, aunque esta información se recolecte y almacene de manera local e independiente en cada dominio. Este requerimiento permite habilitar la ubicuidad en el acceso a esta información, sin incurrir en una centralización o replicación de estos datos. Asimismo, este acceso ubicuo se considera el punto de partida o elemento habilitador de la federación del Meta-OS, ya que constituye un requisito previo esencial para el

funcionamiento de otras funcionalidades colaborativas, como la orquestación descentralizada

- **Orquestación**: cada dominio cuenta con su propia instancia del sistema de orquestación de servicios (HLO y LLO, véase la siguiente sección 4.5.2), por lo que el proceso de orquestación debe seguir el mismo procedimiento con independencia del dominio en el que se lleve a cabo. Esto ocurre gracias a que el bloque de federación permite a los diferentes HLOs establecer comunicaciones directas y relaciones colaborativas, además de habilitar el acceso ubicuo a los datos de contexto clave de todo el continuo.

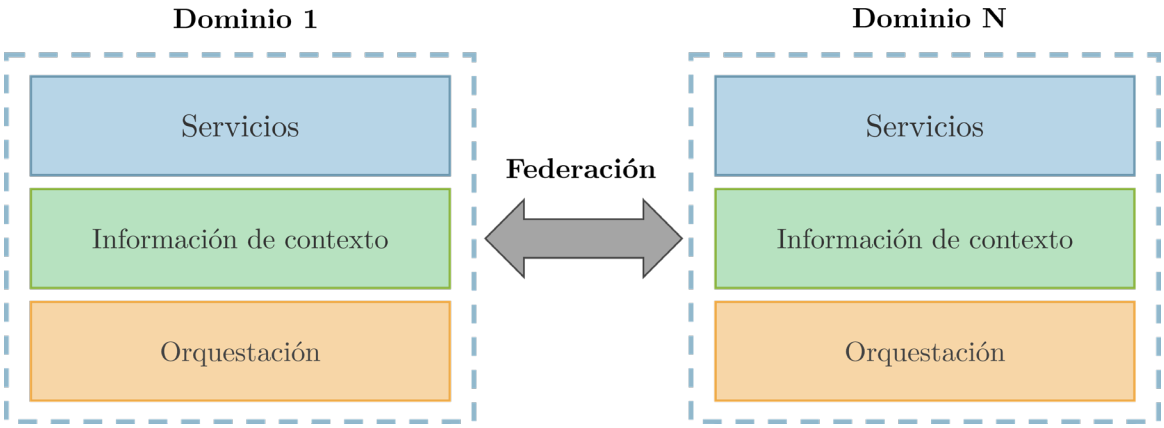


Figura 4.8: Estructura del bloque de federación

4.5.2. Orquestación

Una pieza fundamental que debe incluir necesariamente un Meta-OS para gobernar el continuo de computación es un sistema de despliegue u orquestación de servicios o aplicaciones en el mismo. Este sistema debe ser capaz de tomar decisiones de asignación (*allocation*) de servicios a nodos de computación y llevarlas a cabo de manera tangible (es decir, desplegar el servicio en el nodo elegido). En consecuencia, este proceso de orquestación debe ser realizado de manera inteligente, automática y evidentemente, de forma descentralizada, evitando que este proceso sea realizado constantemente en una serie de nodos previamente seleccionados que actúen como un elemento centralizado, como si se tratara de ejecutar una función que realizara este trabajo en un entorno *serverless*. Asimismo, la arquitectura del bloque de orquestación propuesta es totalmente contraria a dicho paradigma, puesto que en todo momento se tiene constancia de los nodos de computación existentes junto con sus características únicas.

El sistema de orquestación propuesto ha servido de inspiración para la definición de la arquitectura de orquestación para el continuo de computación incluida en el WG-5 de la TF3 dentro de la iniciativa EUCloudEdgeIoT. Esta elección de diseño es objetivamente la más indicada, puesto que, como se mencionó

en la sección 3.4, tanto el doctorando como los directores de esta tesis fueron los impulsores principales de dicha arquitectura. Además, esta decisión refleja dos aspectos fundamentales:

1. **La relevancia del trabajo realizado en esta tesis**, que no solo ha contribuido al desarrollo de la arquitectura, sino que también representa un claro ejemplo de transferencia de conocimiento, alineándose con uno de los principales objetivos de la iniciativa EUCloudEdgeIoT.
2. **El respaldo de esta tesis en estándares ampliamente innovadores**, consolidando su aporte en un campo de investigación en constante evolución.

De acuerdo con esta arquitectura, el sistema de orquestación se divide en dos bloques jerárquicos: el **HLO** y el **LLO**, cuyas descripciones en líneas generales fueron esbozadas en la anterior sección 3.4. El objetivo de ambos módulos es integrar la inteligencia y flexibilidad del Meta-OS en el proceso de asignación de recursos de computación para ejecutar cargas de trabajo en forma de contenedores dentro del continuo. Para este propósito, es necesario emplear algoritmos avanzados de inteligencia artificial (IA) que optimicen ciertos parámetros clave como la latencia o el consumo previsto de recursos, permitiendo una selección inteligente de los recursos más adecuados dentro del continuo. No obstante, el diseño, desarrollo y entrenamiento de estos algoritmos de IA queda fuera del ámbito de esta tesis doctoral por varios motivos. La principal razón es que el desarrollo de un algoritmo de este tipo podría ser en sí mismo una tesis doctoral debido al desafío e innovación que supone su creación. Otro motivo de peso es que actualmente se están desarrollando varias iniciativas en paralelo con este fin, como la propia del proyecto aerOS, o las realizadas en el grupo de investigación SATRD del que forma parte el doctorando [268]. De esta manera, se pretende simplemente integrar alguno de estos algoritmos en la solución desarrollada.

Asimismo, un elemento central en este diseño es la federación entre los distintos dominios que conforman el continuo, ya que permite superar el principio de localidad a nivel de dominio, ampliando el alcance de la solución hasta lograr una visibilidad que engloba el continuo por completo. De esta manera, se pretende que la asignación de recursos se realice de una manera más precisa y eficiente.

En este momento, es conveniente realizar una descripción más detallada de los dos bloques principales, el HLO y el LLO, pero de manera más específica (enfocada a la solución propuesta) que la que se realizó en la sección 3.4:

- Módulo de **orquestación de alto nivel** (*High-Level Orchestrator* — HLO): este módulo tiene un enfoque general para todo el continuo, siendo independiente de los tipos de nodos (en cuanto a su sistema de orquestación de contenedores) que forman un dominio. Cada dominio cuenta con exactamente una única instancia del HLO para cumplir con los principios de federación de dominios y a su vez mantener la

autonomía de los mismos. Esta parte del sistema de orquestación se encarga de tomar la decisión final de despliegue, es decir, decide el nodo en el que se desplegará cada componente del servicio demandado, con independencia del dominio al que pertenezca. Sin embargo, el HLO no despliega realmente los servicios, sino que envía esta decisión al HLO del mismo dominio al que pertenece el nodo seleccionado para que este transmita la orden de despliegue al LLO correspondiente de su dominio local. De esta manera, se establece una relación jerárquica entre el HLO y el LLO, actuando el primero como elemento de control y planificación, mientras que el segundo se limita a ejecutar las órdenes de despliegue emitidas por el primero. Además, se puede afirmar sin problemas que el HLO abstrae al LLO de la complejidad del continuo.

- Componente de **orquestación de bajo nivel** (*Low-Level Orchestrator* — LLO): este componente cuenta con un enfoque completamente específico dependiendo del sistema de gestión de contenedores utilizado por los nodos de un dominio. Por lo tanto, en cada dominio existen tantas instancias de LLOs como diferentes sistemas de gestión de contenedores. En este sentido, las órdenes de despliegue emitidas del HLO del dominio son interpretadas de manera diferente por cada tipo de LLO para plasmarlas de forma tangible (ejecución o eliminación de contenedores) en el nodo elegido. Por ejemplo, si el nodo elegido forma parte de un clúster de K8s, el servicio será desplegado por el LLO en forma de un *Pod*, mientras que, si se trata de un SBC que simplemente tiene instalado Docker, el LLO desplegará un contenedor de Docker.

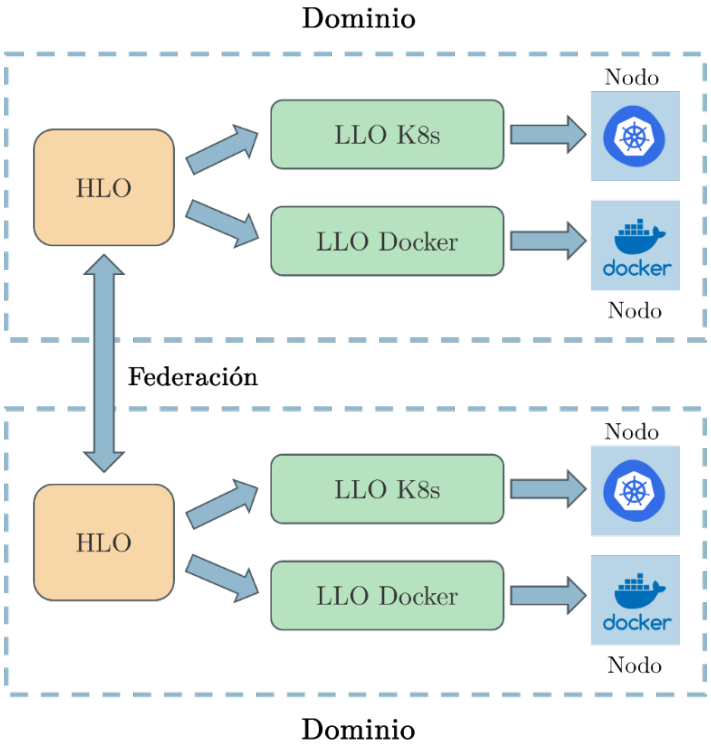


Figura 4.9: Interacción entre los componentes del bloque de orquestación

El proceso de orquestación gestionado por el Meta-OS comienza con la petición de despliegue de un servicio por parte de un usuario final, ya sea a través del portal de gestión (ver punto 4.5.4), utilizando un formulario intuitivo especialmente diseñado para este fin, o bien directamente desde cualquier HLO utilizando su API. Es importante hacer hincapié en que, debido a la naturaleza heterogénea del continuo, no es posible dotar de una completa libertad a los usuarios para expresar los requerimientos del despliegue de dicho servicio. Por lo tanto, esta petición de despliegue debe expresarse a través de la selección y especificación de un conjunto finito de requerimientos predefinidos. Esta serie de requisitos, que pueden incluir parámetros asociados a un acuerdo de nivel de servicio (SLA — *Service Level Agreement*), han sido definidos a partir de la investigación realizada en la primera etapa de esta tesis para que estuvieran alineados con las demás iniciativas relacionadas con el continuo, y que también contaran con una cierta coherencia. Igualmente, se establecen tres modos diferentes de despliegue:

1. **Despliegue totalmente automático:** en este modo los usuarios permiten que el Meta-OS ejerza un total control sobre el proceso de orquestación. De esta manera, el Meta-OS debe elegir la mejor ubicación para el despliegue del servicio de manera inteligente, y usando toda la información que dispone del continuo. Por este motivo, se trata a su vez del modo más interesante a nivel tecnológico y de usuario, pero también del más desafiante.
2. **Despliegue semiautomático:** en este modo los usuarios pueden seleccionar previamente un subconjunto de nodos o dominios para que solamente se tengan en cuenta en el proceso de orquestación, o dotarlos de una mayor prioridad**.** Este modo puede servir como una ayuda previa a los algoritmos de IA en el caso que un usuario no tenga una suficiente confianza en ellos, o incluso si se necesita una mayor precisión en el resultado del proceso de decisión. Asimismo, permite que los usuarios administradores de los dominios o que cuenten con un gran nivel de conocimiento sobre los elementos que componen los dominios puedan tener un mayor nivel de control sobre el proceso. Finalmente, esta intervención humana podría incluso servir para mejorar el entrenamiento de los modelos de IA utilizados.
3. **Despliegue manual:** este tipo de despliegue se trata del más básico, puesto que los usuarios seleccionan un nodo específico en el que desplegar el servicio. En este caso, la petición de despliegue se envía directamente al HLO del dominio elegido para que simplemente envíe la orden de despliegue al LLO correspondiente. No obstante, este modo implica un alto grado de conocimiento de la infraestructura subyacente.

Una vez el usuario final ha definido la petición de despliegue, esta se envía al HLO de un dominio para que sea procesada y se inicie el proceso de orquestación.

Por lo tanto, ha de decidirse en tiempo real el dominio a cuyo HLO va a enviarse esta petición. Desde el primer momento se descartó utilizar un solo dominio para que procesara todas las peticiones, puesto que añadiría un punto de centralización, y potencial punto único de fallo, a una arquitectura con un claro requerimiento de diseño de descentralización. Por ende, se decidió añadir un elemento adicional al sistema de orquestación, el Entrypoint balancer (o balanceador del punto de entrada), que es único para todo el continuo. Este elemento está basado en algoritmos de balanceadores de carga de red del tipo *least connection* y su función principal es distribuir equitativamente entre los diferentes HLOs del continuo las peticiones iniciales de orquestación lanzadas por los usuarios a través del portal de gestión, utilizando la información de contexto del mismo continuo. Esto también es posible debido a que por el propio diseño del HLO y del Meta-OS, cualquier HLO tiene acceso a la misma información clave del continuo y por lo tanto puede realizar el mismo proceso sin que exista diferencia alguna.

Seguidamente, la petición de orquestación es procesada por el HLO que finalmente la recibe. En primer lugar, el HLO realiza un procesado preliminar para adaptarla al modelo de datos común e introducirla en el repositorio común de datos clave del continuo. Seguidamente, se realiza un primer filtrado (cantidad de memoria RAM necesaria, necesidad de un disco SSD o GPU...) para obtener una lista de nodos candidatos que es utilizada como fuente de entrada del algoritmo de IA que se encarga de tomar la decisión de despliegue. Una vez el algoritmo elige el nodo, el HLO prepara con el debido formato la orden final de despliegue y la envía al HLO del dominio al que pertenece el nodo. Finalmente, este HLO envía al LLO necesario de su dominio —dependiendo del sistema de gestión de contenedores del nodo elegido— la orden para que el LLO despliegue el servicio en forma de contenedores. Asimismo, los LLOs son los encargados de monitorizar el estado de los servicios desplegados, puesto que son los elementos que realmente interactúan con los sistemas de gestión de contenedores que los controlan. En resumen, la federación queda patente en 2 niveles: (i) la colaboración entre HLOs para tomar la decisión de despliegue; y (ii) el acceso al repositorio de información clave común.

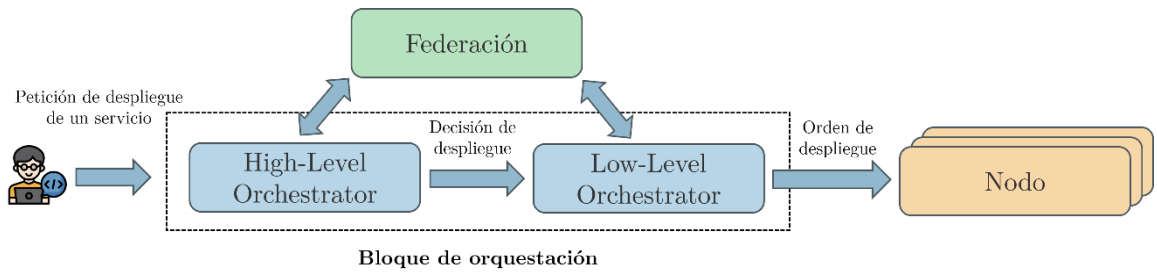


Figura 4.10: Vista general del proceso de orquestación

Por último, el HLO también debe poder gestionar peticiones de reasignación de servicios, es decir, peticiones para que se vuelva a poner en marcha el proceso de orquestación para que un servicio sea desplegado en un nodo diferente. Esta funcionalidad es necesaria debido a que el continuo es un entorno vivo que está en constante cambio, y que además se encuentra presente en la arquitectura de orquestación de EUCEI. Estas peticiones de reorquestación pueden ser emitidas tanto por usuarios finales que no están de acuerdo con el desempeño del servicio en el nodo en el que se está ejecutando o que quieran reaccionar a ciertas notificaciones emitidas por el Meta-OS, como por otros elementos del Meta-OS que lo estimen oportuno.

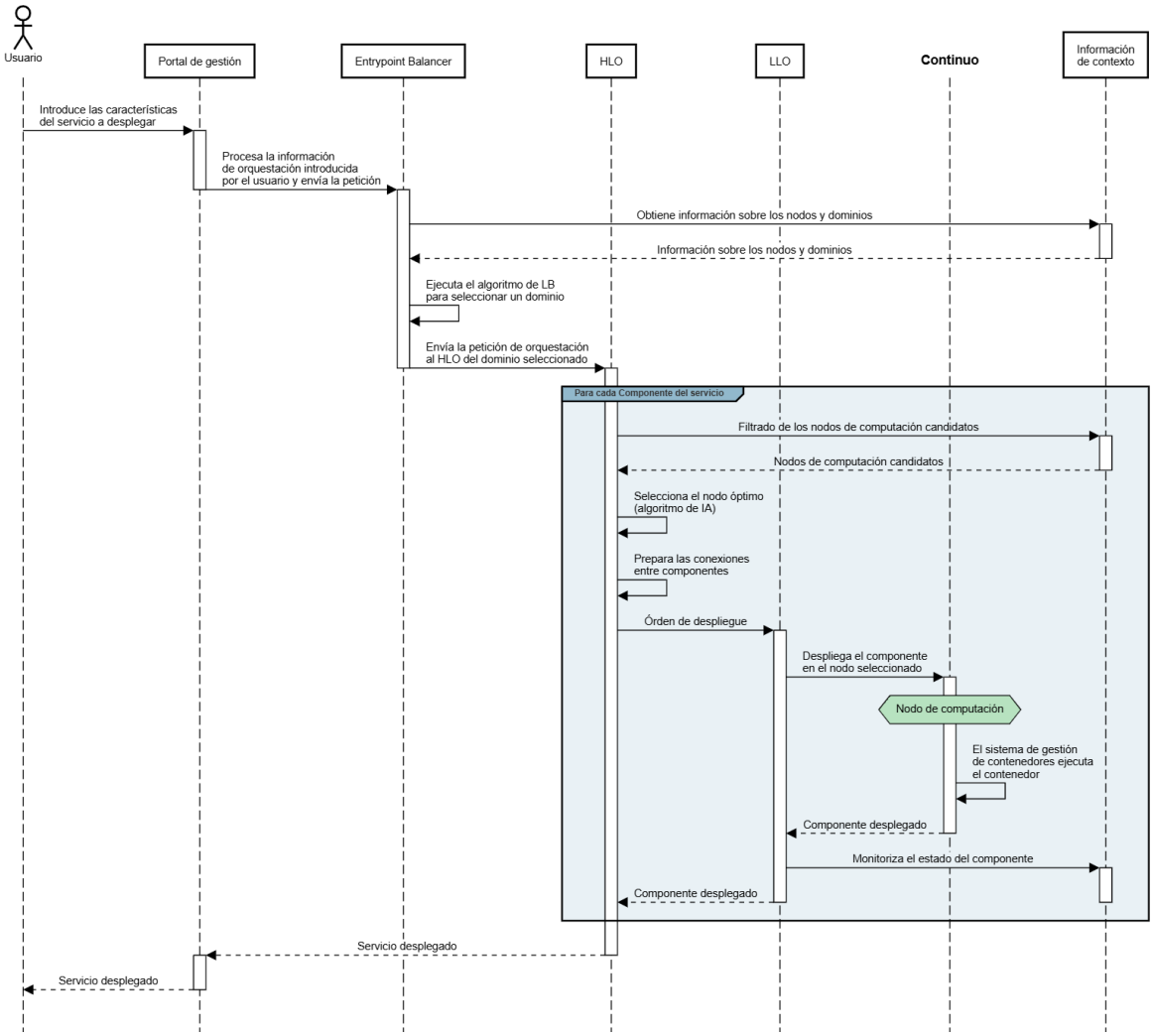


Figura 4.11: Diagrama de flujo general del proceso de orquestación de un servicio

4.5.3. Monitorización y observabilidad

Cualquier sistema informático (e incluso de otro ámbito) no puede ser gestionado debidamente sin que su administrador cuente con el mayor nivel de conocimiento posible sobre el contexto en el que se engloba. Por este motivo, el propósito principal de este bloque, tal y como su propio nombre indica, es dotar de una capa de observabilidad al continuo de computación, de manera que la información heterogénea por naturaleza del continuo se presente de una manera agregada y simplificada tanto al usuario final como a los componentes que conforman los demás bloques del Meta-OS (como el bloque de orquestación), incluso a posibles servicios de usuario desplegados en el continuo que necesiten hacer uso de estos datos para realizar su cometido.

Debida a la propia heterogeneidad que presenta el continuo de computación, el requerimiento indispensable para habilitar este bloque es el establecimiento de un modelo de datos común y transversal donde se incluyan los elementos informacionales más relevantes, y de manera que pueda ser fácilmente usado por los servicios que lo requieran a la que vez que fácilmente entendible por un usuario del Meta-OS. Aunque existen diversos modelos de datos ampliamente establecidos para describir los elementos que conforman un sistema de computación en la nube como TOSCA [269], el doctorando pudo comprobar durante el estudio del estado del arte que no existía una solución ampliamente establecida para el continuo de computación, sobre todo para expresar valores de monitorización del mismo. Por este motivo, se decidió diseñar un modelo de datos para definir el continuo *Cloud-Edge-IoT* adaptado a la solución desarrollada, pero sin perder en ningún momento un criterio de diseño más global para que pueda ser reutilizado por otras iniciativas en el futuro.

En el continuo de computación, la información de contexto realmente la aportan los dominios, nodos de computación, y en última instancia, los servicios desplegados en ellos. Esta información es recogida por diversos elementos de monitorización, dependiendo de la naturaleza de la fuente de la información. Por ejemplo, los dominios son entidades virtuales (constructos definidos con un objetivo práctico dentro del Meta-OS), por lo que su monitorización depende de elementos de monitorización creados específicamente para el Meta-OS concebido en esta tesis. Por otra parte, los nodos de computación y los servicios desplegados en ellos forman parte de elementos tangibles, cuyas características y métricas se pueden medir de manera objetiva. En concreto, existen diferentes soluciones para este fin, incluso cualquier OS de propósito general cuenta con herramientas de monitorización, como *top* en Linux o el Administrador de tareas en Windows. Claros ejemplos son el *Node Exporter* de Prometheus para la monitorización de nodos de computación, o cAdvisor para la monitorización de los recursos consumidos por los contenedores en ejecución.

Estos elementos de monitorización no solamente se encargan de recoger esta información de contexto (compuesta por métricas, control de estados, elementos

virtuales, etc.), sino que han de procesarla para traducirla a un modelo de datos común en un lenguaje en concreto. Finalmente, la información de contexto se indexa en un repositorio único para cada dominio: el gestor de contexto o *context broker*. Este CB actúa como un repositorio agregador de datos clave en el que solo se persiste la información más reciente, ya que su misión es plasmar el estado actual de un sistema. Esto no implica que los datos anteriores se pierdan automáticamente, puesto que existe la opción de persistirlos en un repositorio de datos histórico para aumentar la calidad de la función de observabilidad del Meta-OS, e incluso servir como fuente de datos de potenciales aplicaciones, como podría ser el entrenamiento de un modelo de IA para la mejora del proceso de orquestación, o para la detección de posibles patrones de fallos o de comportamiento del continuo.

Por último, como se ha apuntado en la descripción del bloque de federación, esta información de contexto no se encuentra solamente disponible de manera local y aislada en cada dominio, sino que la arquitectura federada del Meta-OS permite que esta información clave sea compartida para avanzar hacia un sistema totalmente colaborativo, habilitando un nivel de observabilidad global del continuo desde cualquier punto del mismo.

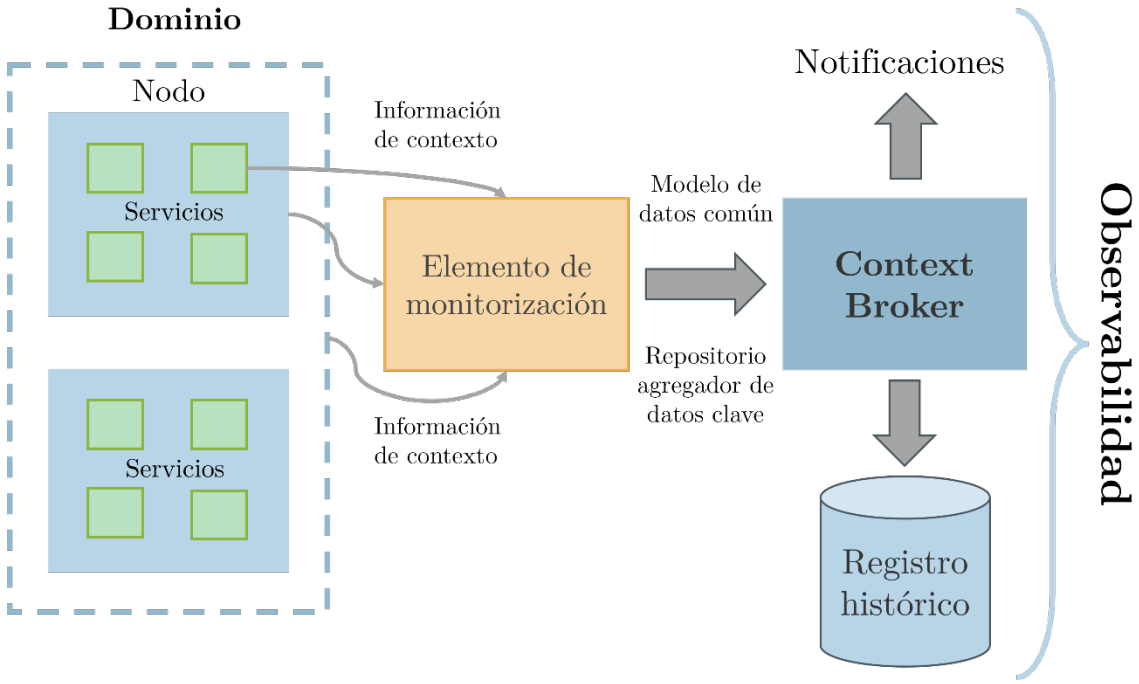


Figura 4.12: Diagrama de bloques del bloque de monitorización y observabilidad

4.5.4. Gestión unificada

Un Meta-OS para gobernar el continuo de computación debe poder ser gestionado por los usuarios finales a través de una interfaz común y reconocible, similar a la de un sistema operativo tradicional. Desde sus inicios, los usuarios de los OS podían interactuar con el sistema mediante un terminal o una línea de comandos, pudiendo realizar tareas como la instalación, desinstalación y ejecución de programas, así como la gestión de usuarios y sus respectivos permisos de acceso. Más tarde, los sistemas operativos evolucionaron para ofrecer a los usuarios una forma más intuitiva y simplificada de interactuar con el sistema, incorporando una interfaz visual (por ejemplo, un escritorio) que opera sobre los procesos internos. Esta interfaz facilita la gestión del sistema mediante *frontends* accesibles y fáciles de usar, mientras que los procesos subyacentes siguen ejecutándose en segundo plano, permaneciendo ocultos para el usuario final.

En el diseño de la solución propuesta se pretende seguir el mismo enfoque adoptado en los sistemas operativos tradicionales, por lo que se ha decidido desarrollar un componente que funcione como un punto de acceso único para que los usuarios finales gestionen el Meta-OS: el portal de gestión unificada del continuo de computación. Este portal, al tener un carácter único como su propia definición indica, solo se va a desplegar en único dominio del continuo, denominado dominio *entrypoint* y cuyas características y propósito se describirán más adelante, lo que facilita la interacción y administración del Meta-OS, y, por ende, del continuo. No obstante, este portal incorpora capacidades de migración, permitiendo su traslado a otro dominio o nodo de computación en respuesta a requerimientos del usuario o a un posible fallo imprevisto, despejando pues toda posible duda de centralización. Este mecanismo evita la pérdida del punto único de gestión, alineándose con principios de alta disponibilidad, en los que la continuidad operativa resulta esencial.

Desde una perspectiva global este enfoque está en total consonancia con los principios fundamentales de diseño de la solución, que priorizan la flexibilidad y la descentralización. Es importante destacar que el concepto de "gestión unificada" no implica que se esté diseñando en base a un modelo computacional centralizado, sino que garantiza un acceso unificado (actuando como un punto único de entrada) a la gestión del continuo sin comprometer la distribución y resiliencia del sistema, además de evitar duplicados innecesarios con el ahorro de recursos que implica, que se hace más patente en entornos *edge*.

Este bloque de gestión se materializa en una aplicación web de página única (SPA, *Single-Page Application*), que actúa como la interfaz principal para la gestión del Meta-OS, mientras que la ejecución de las operaciones correspondientes es realmente realizada en segundo plano por los demás bloques fundamentales del Meta-OS. Por ejemplo, un administrador puede acceder al portal para confirmar la incorporación de un nuevo dominio o de un nuevo nodo al continuo, pero la gestión efectiva de esta incorporación es responsabilidad de los bloques de federación y

monitorización. Además, la interacción entre la Interfaz de Usuario (UI) de este portal y estos servicios básicos está habilitada por el *backend* del propio portal.

Antes de concluir, conviene enfatizar que este portal no introduce nuevas funcionalidades al Meta-OS como tal, sino que aprovecha las funcionalidades ya ofrecidas por el conjunto del mismo para responder a las solicitudes de los usuarios desde un punto de vista funcional. Sin embargo, actúa como el punto de acceso único para la interacción con el continuo, facilitando su administración de manera centralizada y eficiente.

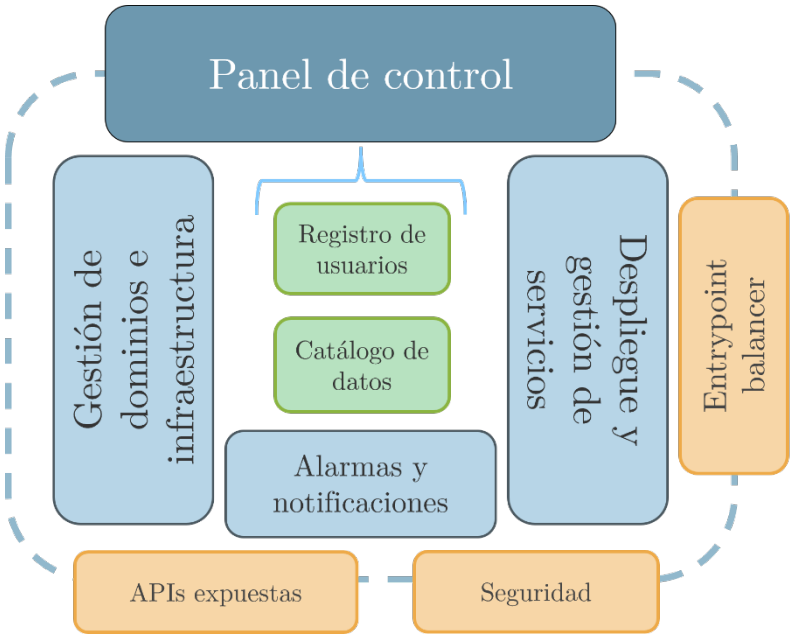


Figura 4.13: Componentes de la plataforma de gestión unificada

4.5.4.1. Dominio *entrypoint*

El concepto de dominio *entrypoint* o de entrada se ha introducido previamente mientras se describía el bloque de gestión unificada del Meta-OS. Se trata de un dominio diferente a los demás en cuanto a que contiene los elementos necesarios para la gestión del continuo y del propio Meta-OS, además de ejercer como desencadenante de ciertos procesos, pero no cuenta con otros elementos diferenciadores o funcionalidades únicas. En resumen, en el dominio *entrypoint* se despliegan de manera única el portal de gestión del continuo, la herramienta de gestión de los usuarios que pueden acceder a este portal, y el elemento Entrypoint balancer, ya que recibe las peticiones de orquestación directamente del portal.

De igual manera que en el proceso de diseño y establecimiento de dominios, la elección del dominio *entrypoint* es competencia exclusiva de los administradores o propietarios de la infraestructura para que sigan contando con una total libertad de diseño del continuo, puesto que estos con total seguridad disponen de un mayor nivel de conocimiento sobre los elementos subyacentes del mismo. Aunque exista

dicha libertad de diseño, esta elección tiene un impacto en términos operativos tanto en el Meta-OS como en el continuo. Usualmente se elige al dominio cuyos nodos tienen una mayor capacidad de computación, o que alguno de ellos cuente con alguna dirección IP pública o sea directamente alcanzable por los demás dominios. Puesto que estas características suelen coincidir con los dominios puramente *cloud* o incluso los de carácter *edge* más potentes, se podría afirmar que existe una tendencia a identificar un dominio *cloud* como el dominio *entrypoint.*

Capítulo 5

Integración de la solución

5.1.Introducción

En el capítulo anterior se describieron con gran nivel de detalle todos los aspectos relacionados con el diseño de la solución propuesta para cumplir con el objetivo principal de esta tesis doctoral, que recordemos se trata del diseño y la materialización de los componentes claves que habilitan un Meta-OS para el continuo de computación *Cloud-Edge-IoT*. Estos componentes se describieron a alto nivel, o, dicho de otro modo, su funcionalidad se describió de una manera general utilizando unos esquemas basados en una estructura de bloques. Sin embargo, este contenido no fue el único que contempla el anterior capítulo, ya que también se esbozó el proceso de diseño previo al desarrollo de la solución mediante la descripción de los requisitos y de la metodología seguida.

El principal propósito de este capítulo es ampliar el contenido introducido en el anterior, ahondando tanto en su estructura interna (componentes, interacción entre ellos y otros módulos del Meta-OS), como en las herramientas y tecnologías utilizadas para su desarrollo e implementación real. Por lo tanto, el contenido de este capítulo es de carácter profundamente técnico, mostrando un nivel de detalle superior al mostrado hasta este punto de la tesis, excepto en algunas secciones del capítulo del estado del arte, puesto que es necesario recordar que esta tesis doctoral se enmarca en un área de conocimiento técnica en esencia, como es la Ingeniería de Telecomunicaciones. Es más, la influencia de la investigación realizada en la primera etapa de la tesis queda suficientemente plasmada en el contenido de este capítulo.

Por último, la estructura del capítulo es clara, ya que está directamente relacionada con los bloques mencionados en el párrafo anterior y que se expusieron en el apartado 4.5. Además, se ha decido añadir dos secciones adicionales que son transversales al desarrollo de dichos bloques: la descripción de la estrategia de empaquetado común a todos los desarrollos realizados y la enumeración de elementos adicionales o complementarios que pueden presentarse en un Meta-OS, pero que no forman parte del núcleo principal de esta tesis.

5.2.Monitorización y observabilidad

5.2.1. Modelo de datos: ontología para modelar el continuo

El continuo *Cloud-Edge-IoT* que ha de gestionar un Meta-OS representa un paradigma de computación distribuida en la que los datos fluyen sin diferencias tangibles desde los dispositivos IoT, localizados en el extremo de la topología, hasta infraestructura de computación en la nube centralizada. Esta alta heterogeneidad y complejidad inherente al continuo de computación necesita ser modelada en una ontología o modelo de datos que sea lo más simple y homogénea posible, de manera que pueda ser a la vez entendible por los usuarios finales del Meta-OS y eficiente para las comunicaciones entre máquinas (también llamadas *machine-to-machine communications* — M2M). Esta es, pues, una funcionalidad esencial cubierta en esta tesis.

Tal y como se justificó en el anterior apartado 4.5.3, se decidió desarrollar un modelo de datos completamente nuevo y adaptado al continuo de computación que se concibe en el contexto de esta tesis debido a la poca cantidad de opciones consolidadas existentes, pero sin perder un espíritu generalizador o envolvente con el objetivo de que pueda ser estandarizado o ser usado por otros proyectos en el futuro, ya que la visión del continuo imprimada en esta tesis está alineada con las iniciativas referentes en este ámbito de conocimiento. Por este motivo también se decidió usar el inglés para definir las entidades y atributos de este modelo de datos,

como se podrá comprobar más adelante. Aunque la elaboración de este modelo de datos se ha realizado desde cero, es necesario admitir que algunas partes de este modelo de datos han sido inspiradas por otras iniciativas como FOAF, OASIS TOSCA o FIWARE Smart Data Models, pero siempre adaptándolas al contexto del continuo de computación.

En esencia, este modelo de datos tiene la intención de encapsular los conceptos esenciales, relaciones, interdependencias y propiedades relevantes en cuanto a la gestión de datos, computación y orquestación del continuo. Por ende, ha sido diseñado teniendo en cuenta tres pilares fundamentales del Meta-OS propuesto (precisamente aquellos principalmente abordados en esta tesis): (i) Federación de dominios, (ii) Gestión unificada del continuo, y (iii) Orquestación descentralizada.

Sin más dilación, en la siguiente Figura 5.1 se puede observar el modelo de datos completo que se ha diseñado (en la Figura A.1 del Apéndice A se muestra de forma ampliada para mejorar su legibilidad). Para crear este diagrama se ha decidido utilizar una analogía con un modelo de bases de datos relacionales (como SQL, por ejemplo), en el que las diferentes entidades se muestran como tablas con una serie de atributos. Asimismo, las relaciones entre entidades se muestran como enlaces relacionales entre sus atributos.

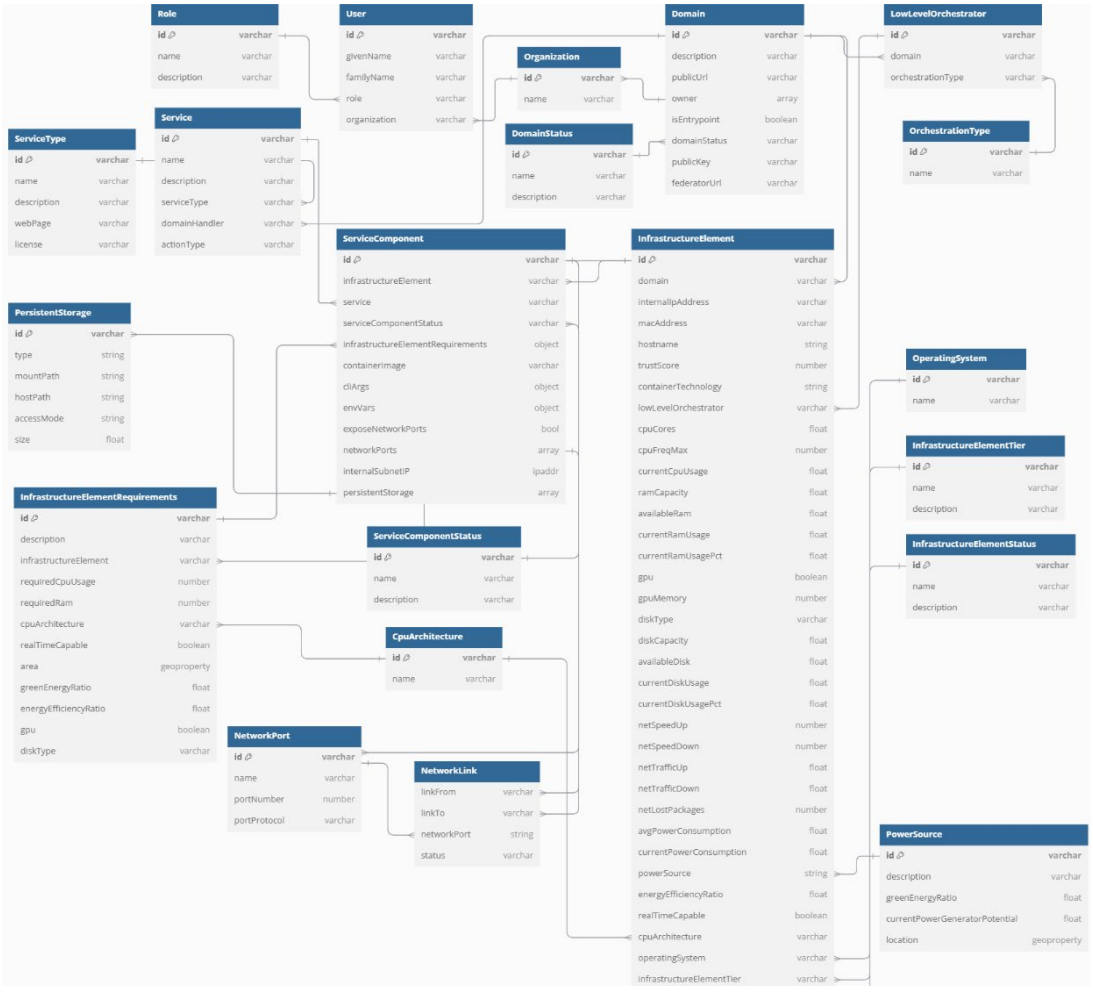


Figura 5.1: Modelo de datos del continuo de computación CEI

Gestión administrativa: usuarios, roles y organizaciones

Este bloque del modelo de datos tiene la intención de representar la parte más humana del continuo, es decir, la relación entre los actores humanos (los usuarios del Meta-OS) y los recursos de computación y los servicios desplegados en ellos. Se puede afirmar que esta es la parte menos innovadora del modelo de datos, puesto que la definición de usuarios, su clasificación u organización, junto con la definición y asignación de diferentes roles asociados a diferentes permisos ha estado presente en todos los sistemas de computación desde su misma creación. Por este motivo, se decidió reutilizar y adaptar el concepto *Person* de la ontología FOAF (Friend Of A Friend), una de las ontologías de referencia a la hora de definir usuarios en un sistema [270]. Cada usuario del Meta-OS (o incluso de un continuo de computación) es miembro de una o varias organizaciones y tiene asignado una serie de roles con sus permisos asociados. Además, los dominios del continuo pertenecen a las organizaciones en lugar de a los usuarios a nivel personal.

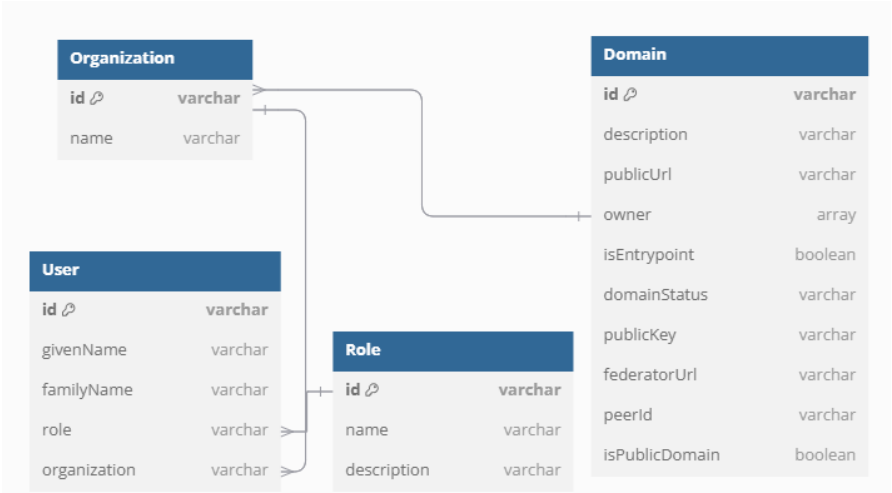


Figura 5.2: Bloque de gestión administrativa del modelo de datos

Infraestructura

El bloque de infraestructura del modelo de datos permite representar los recursos de computación que forman parte del continuo con un alto nivel de detalle, además de su agrupamiento en los diferentes dominios para formar un continuo de computación *Cloud-Edge-IoT* completo. Por este motivo, la entidad *ComputingNode*, que representa los nodos de computación del continuo, emerge como elemento central no solo de este bloque de infraestructura, sino del modelo de datos por completo.

Antes de continuar, es importante apuntar que, tal y como se puede comprobar en los esquemas, se ha decidido modelar como entidades independientes aquellos atributos con un mayor nivel de importancia (por ejemplo, el OS o el estado de un nodo), y que se pueden clasificar y modelar a su vez (contando con

sus propios atributos), por lo que se establece una relación de dependencia entre entidades.

Los atributos de la entidad *ComputingNode* pueden clasificarse en tres grupos en base a su naturaleza:

- **Atributos descriptivos:** son aquellos atributos que describen las características de un nodo de computación a diferentes niveles. En cuanto al nivel *hardware*, se incluyen atributos como la arquitectura de su CPU (a su vez modelada en una entidad independiente), si cuenta con una GPU, el tipo de disco duro o la cantidad de memoria RAM. Por otra parte, también se incluyen características lógicas o de infraestructura como la dirección IP interna, el clúster de K8s al que pertenece —si es el caso— o su localización física en coordenadas. Además, también es importante indicar atributos directamente relacionados con el Meta-OS o el continuo de computación, como el dominio al que pertenece el nodo o el LLO que lo gestiona a nivel de orquestación (que al mismo tiempo indica el sistema de gestión de contenedores que se ha instalado en el nodo).

  Por lo tanto, estos atributos se prevé que sean mayoritariamente estáticos, o susceptibles a unos pocos cambios durante su ciclo de vida.
- **Atributos dinámicos o métricas:** estos atributos pretenden plasmar el estado actual del nodo de computación —monitorización—, mostrando los valores actuales de consumo de recursos de la máquina. Claros ejemplos son la cantidad de memoria RAM utilizada, el espacio libre en disco o el nivel de carga de la CPU.
- **Atributo de estado:** este atributo único (*computingNodeStatus*) tiene como propósito plasmar el estado actual del nodo de computación. Los valores definidos son: *Ready, Overload, Unsecure* y *Untrusted*.

La siguiente entidad por orden de importancia de este bloque se trata de la entidad *Domain*, la cual representa la entidad virtual o administrativa llamada dominio que actúa como elemento agrupador de entidades *ComputingNode*. Esta entidad cuenta con atributos como su URL de acceso principal o un *flag* (atributo *booleano*) para indicar si se trata del dominio único *entrypoint*, además de otras características importantes del Meta-OS que se irán describiendo en las siguientes secciones a medida que se vayan detallando los bloques funcionales relacionados con ellas. Esto es debido a que, como ya se ha apuntado anteriormente, este modelo de datos actúa como un importante bloque habilitador de la solución propuesta. Por último, también cuenta con un atributo de estado que intenta representar el estado global del dominio, cuyos valores son: *Functional, Preliminary, Removed* y *Untrusted*.

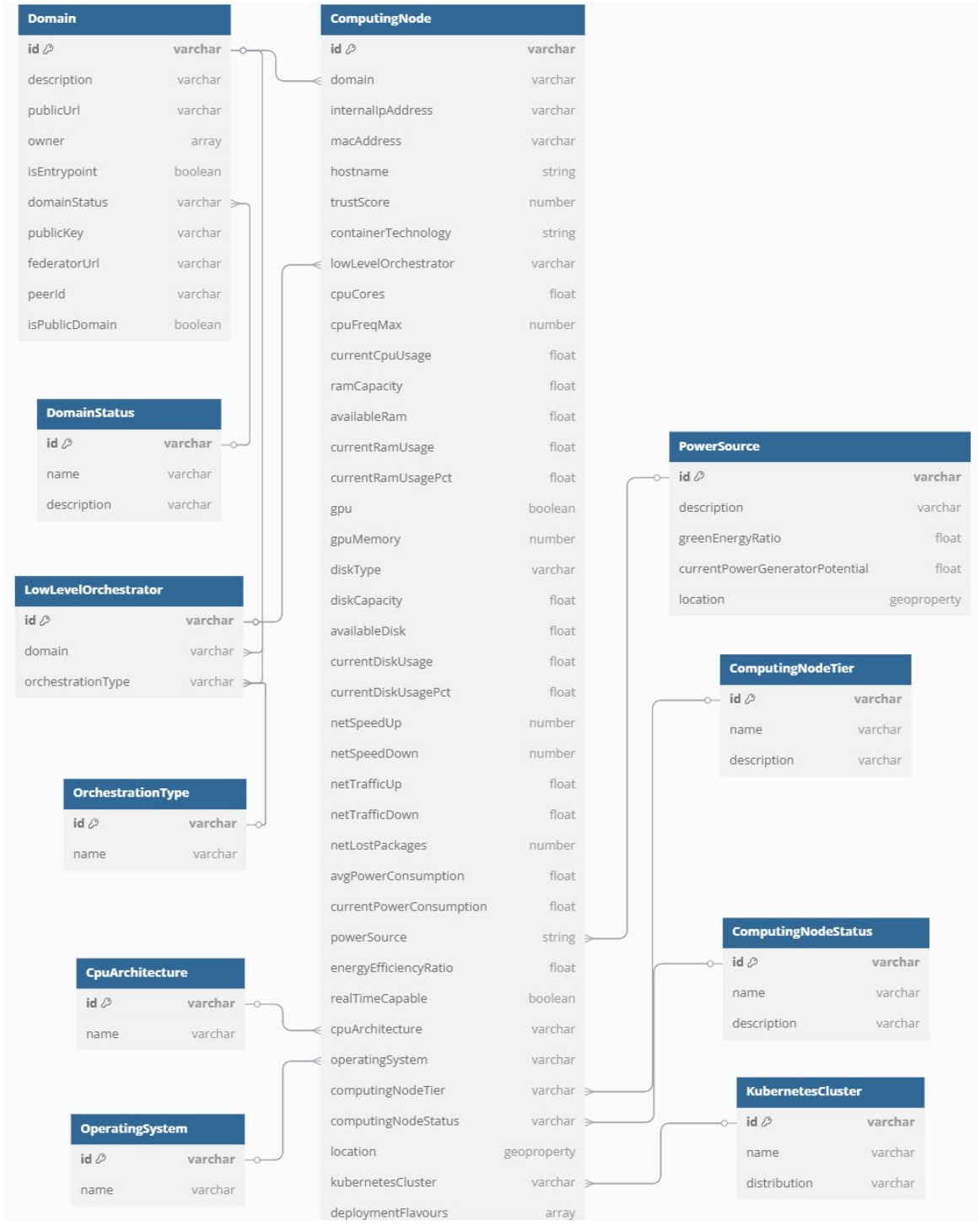


Figura 5.3: Bloque de infraestructura del modelo de datos

Orquestación de servicios

El bloque de servicios tiene como propósito plasmar (y ejercer como soporte de) los múltiples procesos internos que conforman el acto global de orquestación diseñado en este Meta-OS, así como modelar los servicios de usuario desplegados en el continuo de computación.

La entidad *Service* ha sido concebida como una capa conceptual a modo de agrupamiento que actúa sobre la entidad *ServiceComponent*, estableciendo de esta manera una relación de 1:N entre servicios y los componentes que los conforman, siendo completamente equivalente a la relación anteriormente descrita entre dominios y nodos de computación. Esta decisión de diseño tiene como propósito ser compatible con el patrón de diseño de microservicios ampliamente establecido en entornos tanto *cloud-native* como *edge-native*, además de con aplicaciones con arquitecturas más tradicionales o monolíticas en las que el servicio o aplicación está formada por un elemento principal junto con elementos complementarios. Por ejemplo, en el caso de Orion-LD, el servicio estaría formado por dos componentes: el propio *broker* Orion-LD y una base de datos MongoDB. No obstante, es necesario recordar que en el proceso de orquestación diseñado se despliegan servicios completos, compuestos por diferentes componentes que corresponden directamente con los contenedores que se van a desplegar, por lo que realmente el proceso de orquestación y despliegue actúa a nivel de componente (por tanto, tal y como se ha mencionado, la agrupación *Service* no es más que una abstracción). En consecuencia, la entidad *ServiceComponent* se establece como el elemento central del bloque de orquestación del Meta-OS desarrollado.

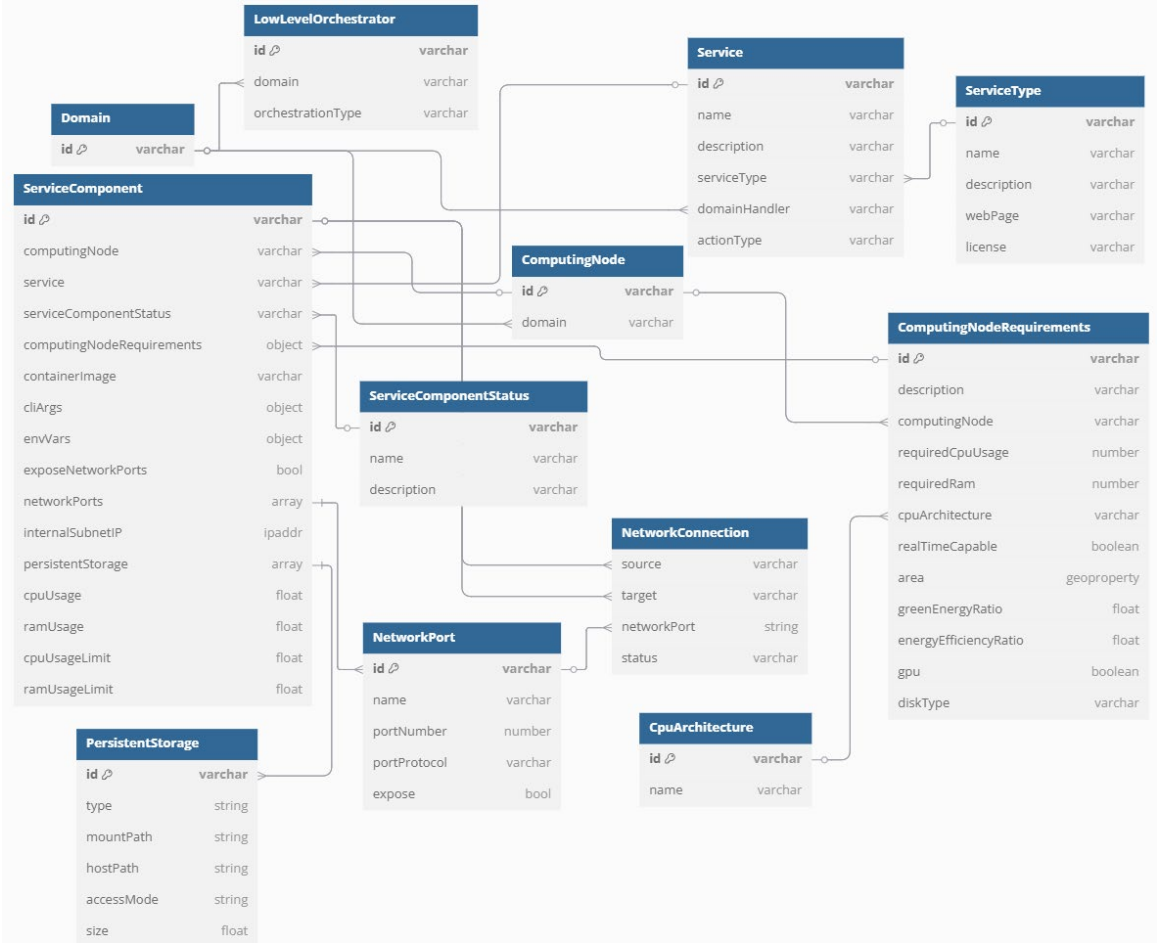


Figura 5.4: Bloque de servicios del modelo de datos

Los atributos de la entidad *ServiceComponent*, al igual que en la entidad *ComputingNode*, se pueden clasificar en tres grupos en base a su naturaleza:

- **Atributos descriptivos o de orquestación:** son aquellos atributos que describen las características del componente desplegado por un usuario. Se pueden clasificar a su vez en dos grupos: i) características introducidas por el usuario para iniciar el proceso de orquestación (servicio al que corresponden, límite de uso de recursos, imagen de contenedor, variables de entorno, puertos de red que exponen, conexiones con otros servicios, persistencia...); y (ii) características introducidas por el proceso de orquestación (nodo seleccionado para su despliegue, requerimientos de orquestación – en este caso se crea una nueva instancia de la entidad *ComputingNodeRequirements* a partir de los requerimientos introducidos por el usuario).
- **Atributos dinámicos o métricas:** estos atributos pretenden plasmar el consumo de recursos de los contenedores en el nodo en el que se están ejecutando. Actualmente solo incluyen el consumo de memoria RAM y el nivel de carga de la CPU.
- **Atributo de estado:** este atributo único (*serviceComponentStatus*) tiene como propósito plasmar el estado actual del componente desplegado, por lo que se establece como la fuente única de verdad (SSOT – *Single Source Of Truth*) que es alimentada por la funcionalidad de LCM de servicios completa que aporta el Meta-OS, y que se describirá con un amplio nivel de detalles en la sección 5.4.2.1. Los valores definidos son: *Deployed*, *Starting, Running, Updating, Overload, Failed, Migrating, Locating, Removing* y *Finished*.

En cuanto a la entidad agregadora *Service*, incluye atributos para plasmar el dominio cuyo HLO ha sido encargado de ejecutar el proceso de su orquestación (recordemos que debido a la naturaleza federada de este Meta-OS no importa el dominio en el que se ejecuta el proceso de selección del nodo de despliegue), el estado del proceso interno que se está llevando a cabo o se ha llevado a cabo en el HLO (atributo *actionType*, cuyos valores pueden ser *Deploying, Removing, Deployed* o *Finished*), e incluso un intento de catalogar los servicios más comunes a través de la entidad *ServiceType* (nombre, descripción, licencia, página web con su documentación, etcétera).

Finalmente, antes de dar por concluida esta sección, resulta relevante destacar que el diseño y desarrollo del modelo de datos se ha abordado desde un enfoque descriptivo y teórico, describiendo las diferentes entidades junto con sus atributos y las relaciones entre ellos, que lo conforman.

Este enfoque garantiza que el modelo de datos mantenga un alto grado de flexibilidad, permitiendo su implementación en cualquier herramienta, lenguaje de programación o sistema de modelado de datos, sin imponer restricciones específicas. En el caso concreto de esta tesis, se ha utilizado principalmente NGSI-LD como

lenguaje de implementación del modelo de datos, como se detallará en la siguiente sección.

5.2.2. Gestor de contexto y materialización del modelo de datos

Como se explicó en la sección 4.5.3, el elemento principal de este bloque que actúa como agregador de datos clave es el gestor de contexto o *context broker*, por lo que la tarea inicial para implementar este bloque funcional del Meta-OS consiste en la elección de un CB que cuente con los requerimientos anteriormente definidos y que sea capaz de soportar la gran carga funcional y de rendimiento que implica el hecho de ser un elemento esencial del Meta-OS. El doctorando ha decidido usar el CB Orion-LD de FIWARE (también descrito en el apartado 2.4.1) por dos motivos principales: (i) esta herramienta fue nombrada habilitador oficial de la infraestructura digital dentro del programa Constructing Europe Facilities (CEF) de la Comisión Europea [271]; (ii) debido a sus amplios conocimientos sobre esta herramienta y el completo ecosistema FIWARE, puesto que el doctorando obtuvo recientemente la certificación de experto en FIWARE (*FIWARE Expert*) [272]. Además, durante el desarrollo del proyecto aerOS, el doctorando trabajó en la mejora y adaptación de Orion-LD para su uso en entornos distribuidos y no tan propios del IoT, especialmente en la parte de peticiones distribuidas o federación entre diversas instancias de este gestor de contexto. Concretamente, se mantuvo una colaboración fluida con la FIWARE Foundation y el equipo desarrollador principal de Orion-LD, proponiendo nuevas funcionalidades y mejoras, así como testeando las mismas directamente en los casos de uso en los que se validó el sistema desarrollado. Esta interacción se puede comprobar de manera pública mediante los diversos *issues* de GitHub en los que se plasma todo este proceso.

Orion-LD implementa la especificación NGSI-LD, en la que se utiliza JSON-LD como lenguaje para definir los datos de contexto, que a su vez se agrupan en entidades. Por este motivo, el modelo de datos del continuo debe implementarse utilizando este lenguaje, transformando cada concepto de este modelo en una entidad de NGSI-LD con un *type* único, de la que pueden existir varias instancias con su identificador único (ID). Las entidades cuentan con una serie de atributos, que pueden indicar un valor (tipo *Property*, como la cantidad de memoria RAM disponible en un nodo) o una relación que apunta al ID de otra entidad (tipo *Relationship*, como el dominio al que pertenece un nodo, usando como valor en este caso el ID de la entidad del dominio en cuestión). Asimismo, Orion-LD introduce varios metadatos por defecto (técnicamente llamados subatributos) que consisten en marcas temporales (en formato ISO 8601) para reflejar en qué momento exacto se ha creado y modificado la entidad o cualquiera de sus atributos.

En el Meta-OS propuesto, en los gestores de contexto se almacenan entidades que mapean con los elementos del continuo (por ejemplo, un dominio se representa en una entidad del tipo *Domain*), reflejando el estado actual del continuo en todo

momento. De esta manera se logra aprovechar el propósito inicial con el que fueron diseñados los gestores de contexto, pero moviéndolos a un ámbito diferente para el que fueron pensados inicialmente mostrando a su vez su flexibilidad como el trabajo realizado en esta tesis para demostrar la adaptabilidad de tecnologías ampliamente establecidas en otros verticales a paradigmas modernos.

A modo de ejemplo ilustrativo, los nodos de computación se representan en una entidad cuyo *type* es *ComputingNode*, y cada uno de los nodos tiene un identificador único que sigue el patrón *urn:ngsi-ld:ComputingNode:{Dominio}:{DirecciónMAC}*. Mientras que sus atributos que representan métricas puras (*availableRam*, *currentCpuUsage*...) siempre son del tipo *Property*, otros atributos que representan características propias pueden ser de ambos tipos. De esta manera, entidades como la dirección MAC o la dirección IP interna que son valores únicos y con un formato predefinido se representan en entidades de tipo *Property*, mientras que atributos cuyo valor es una serie de valores previamente modelados como la arquitectura de CPU o el estado, se representan en entidades de tipo *Relationship*. Finalmente, los atributos que directamente apuntan a otra entidad clave del continuo siempre son del tipo *Relationship*, como *domain* o *lowLevelOrchestrator*.

Además, NGSI-LD ofrece tres modos de representación de las entidades basados en el nivel de detalles de su representación: *simplified, concise* y *normalized*. Mientras que en el primer modo las entidades se representan solamente con el valor de sus atributos (se podría decir que es un modelo clave-valor), en los dos modos restantes los atributos se expanden hasta un objeto JSON en el que se incluye el tipo de entidad junto con sus subatributos adicionales. En la siguiente figura, aunque se haya recortado cierto contenido para priorizar una visualización comparativa, se puede apreciar las diferencias entre el primer y el último modo, incluyendo los metadatos introducidos por el CB por defecto.

```
{
  "id": "urn:ngsi-ld:ComputingNode:Production:00155dc09474"
  "type": "ComputingNode",
  "domain": "urn:ngsi-ld:Domain:Production",
  "hostname": "continuum-worker-1",
  "internalIpAddress": "192.168.1.35",
  "macAddress": "00:15:5d:c0:94:74",
  "trustScore": 0.58,
  "containerTechnology": "Kubernetes",
  "lowLevelOrchestrator": "urn:ngsi-ld:LowLevelOrchestrator
  "cpuCores": 4,
  "currentCpuUsage": 20,
  "ramCapacity": 8192,
  "availableRam": 5100,
  "currentRamUsage": 3092,
  "diskCapacity": 65536,
  "availableDisk": 30585,
  "currentDiskUsage": 34951,
  "avgPowerConsumption": 100,
  "currentPowerConsumption": 95,
  "realTimeCapable": true,
  "cpuArchitecture": "urn:ngsi-ld:CpuArchitecture:x64",
  "operatingSystem": "urn:ngsi-ld:OperatingSystem:Linux",
  "computingNodeTier": "urn:ngsi-ld:ComputingNodeTier:Edge"
  "computingNodeStatus": "urn:ngsi-ld:ComputingNodeStatus:R
  "location": {
    "type": "Point",
    "coordinates": [
      -0.3411743020092746,
      39.48103451588121
    ]
  }
}
```

```
{
  "id": "urn:ngsi-ld:ComputingNode:Production:00155dc09474",
  "type": "ComputingNode",
  "createdAt": "2024-12-17T08:00:02.106Z",
  "modifiedAt": "2025-02-08T16:57:25.032Z",
  "domain": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2024-12-17T08:00:02.106Z",
    "type": "Relationship",
    "object": "urn:ngsi-ld:Domain:Production"
  },
  "hostname": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2024-12-17T08:00:02.106Z",
    "type": "Property",
    "value": "continuum-worker-1"
  },
  "lowLevelOrchestrator": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2024-12-17T08:00:02.106Z",
    "type": "Relationship",
    "object": "urn:ngsi-ld:LowLevelOrchestrator:Production:K
  },
  "cpuCores": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2024-12-17T08:00:02.106Z",
    "value": 4
  },
  "currentCpuUsage": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2025-02-08T16:57:25.032Z",
    "value": 21,
    "unitCode": "Percentage (%)"
  },
  "availableRam": {
    "createdAt": "2024-12-17T08:00:02.106Z",
    "modifiedAt": "2025-02-08T16:57:25.032Z",
    "value": 5100,
    "unitCode": "Megabytes (MB)"
  }
```


Figura 5.5: Ejemplos de una entidad NGSI-LD en formato *simplified* y *normalized*

Orion-LD, como *context broker* NGSI-LD, no solo permite ofrecer y almacenar la información de contexto, sino que también permite la creación de suscripciones para monitorizar los atributos clave de las entidades con el propósito final de emitir notificaciones cuando se cumplen ciertas reglas predefinidas. Por lo tanto, el CB sirve también como fuente de emisión de notificaciones o alerta a otros componentes del Meta-OS o incluso a usuarios finales [273]. Un ejemplo sería el aviso de que un cierto nodo ha superado un límite de uso de memoria RAM, que podría desencadenar un proceso de reorquestación de un servicio para aligerar la carga del nodo, o simplemente mostrar este aviso en el portal de gestión para que un administrador reaccione acordemente.

A modo de conclusión, cabe recordar que los gestores de contexto solo persisten por defecto el valor más reciente de los atributos de las entidades, utilizando en el caso de Orion-LD una instancia de la base de datos no relacional de MongoDB [274]. No obstante, este hecho no significa que no se pueda crear un registro histórico de los valores de los atributos de estas entidades, ya que existen diversas herramientas en el ecosistema FIWARE para dicho propósito, y basándose en la propia experiencia del doctorado en otros desarrollos, incluso se podría

desarrollar un sistema propio de manera bastante sencilla que a su vez esté basado en las propias suscripciones de NGSI-LD.

Como se ha apuntado, FIWARE proporciona herramientas *plug-and-play* para persistir esta información histórica: utilizando bases de datos temporales junto con una API que implementa un sistema de peticiones en base a parámetros temporales (Mintaka junto con TimescaleDB), bases de datos relacionales (PostgreSQL) o no relacionales (Elasticsearch o incluso el mismo MongoDB). Esta diversidad en el uso de múltiples paradigmas de almacenamiento o persistencia de las entidades justifica la afirmación hecha en la sección anterior, en la que se defendía la flexibilidad y adaptabilidad de este modelo de datos para ser usado en cualquier tipo de entorno tecnológico.

5.2.3. Monitorización de los elementos clave

Una vez detallado el modelo de datos diseñado para permitir la monitorización, y exponer la herramienta tecnológica para soportar la actualización de dichos valores, conviene mencionar las estrategias existentes en el Meta-OS para conseguir, efectivamente, introducir dichos valores. Es decir, obtener la información de la fuente primaria (por ejemplo, valores del sistema operativo de un *ComputingNode*) e informarlos al *context broker* siguiendo dicho esquema de datos.

En este punto es importante diferenciar entre las entidades del modelo de datos que representan elementos reales y tangibles del continuo, y entidades conceptuales con un propósito administrativo u organizativo. En palabras más coloquiales, se trata de identificar los elementos que han de ser monitorizados en tiempo real mediante la recolección de métricas clave, aunque a nivel de la implementación en el estándar NGSI-LD no implique diferencia alguna, dado que todas las entidades siguen teniendo el mismo formato.

Estas entidades clave son el *ComputingNode*, cuyos valores y métricas deben recolectarse directamente de nodos de computación, interactuando con el sistema operativo subyacente o incluso algunos han de ser introducidos manualmente por los usuarios instaladores del Meta-OS; y el *ServiceComponent*, puesto que la entidad *Service* recordemos que se trata de una entidad administrativa que engloba una serie de *ServiceComponents*. Las métricas de este último deben obtenerse de los sistemas de gestión de contenedores en lugar de descender hasta el mismo proceso del sistema correspondiente al contenedor.

Debido a estos requerimientos, queda patente la necesidad de desarrollar dos elementos de monitorización diferentes, que a su vez actualizarán constantemente la entidad correspondiente en el gestor de contexto para dotar de una capa de observabilidad al continuo de computación.

Además de estas métricas, es completamente necesario monitorizar el estado de los anteriores elementos clave, pero sumando a ellos la entidad dominio. Este

tipo de monitorización no es tan directa u objetiva como la anterior, siendo más bien subjetiva, puesto que no existe una forma única de "medir" el estado de un componente del continuo. Por ejemplo, la cantidad de espacio libre en el disco de una máquina se trata de una métrica objetiva, pero un estado de *Overload* o *Untrusted* de un nodo no tiene una percepción relativa con una situación real, sino que ha de ser "informado" por otro elemento específico customizado del Meta-OS. En el caso de un dominio, al tratarse de una entidad virtual en sí, no es realizada directamente por un elemento de monitorización, sino que diferentes componentes del Meta-OS pueden detectar un fallo en el funcionamiento de alguno de los elementos que componen un dominio y actualizar su estado en consecuencia, para que otro elemento del Meta-OS o un usuario pueda reaccionar acordemente.

5.2.3.1. Monitorización de los nodos

A continuación, se van a describir las dos alternativas desarrolladas para la monitorización de los nodos de computación. Pero, en primer lugar, es conveniente apuntar las características comunes con las que cuentan ambas:

- Estos elementos tienen una ejecución periódica y configurable durante su ejecución, que por defecto es de 5 segundos. Este tiempo de ejecución es el que define la frecuencia con la que se actualizará la información de monitorización de los nodos de computación.
- En cada nodo de computación ha de desplegarse solamente una única instancia de una de las dos soluciones, por lo que la ejecución de ambas en el mismo nodo queda descartada al implicar un consumo innecesario de recursos. En el caso de un clúster de K8s, sería recomendable el uso de una única solución.
- Uso del modelo de datos definido y compatibilidad con la API de NGSI-LD para la actualización de los atributos de únicamente la entidad de tipo *ComputingNode* asociada con el nodo en el que se despliega.
- Se encargan de la creación de la entidad de tipo *ComputingNode* correspondiente en el CB si no ha sido previamente creada, especificando el valor de los atributos considerados estáticos o descriptivos relacionados con sus características (núcleos de la CPU, dirección IP interna, dirección MAC, tipo de disco duro, etc.).
- Solamente se actualiza el valor de los atributos considerados dinámicos, es decir, relacionados con el consumo de recursos (uso de memoria, CPU, almacenamiento...), para evitar sobrecargar la red con el envío de información repetitiva.

Módulo Self-awareness

Aunque existen elementos del *stack* de FIWARE (como el IoT Agent [275]) para este propósito, se descartó su uso debido a la experiencia previa del doctorando en el desarrollo de agentes NGSI totalmente adaptados para las diversas fuentes de datos y objetivos concreto de los proyectos PIXEL y ASSIST-IoT, mayoritariamente desarrollados en Python [276]. El funcionamiento de estos agentes es simple: recolectan datos en crudo de diferentes fuentes que son accesibles mediante diferentes medios (APIs, *brokers* MQTT, páginas web estáticas, protocolos industriales, etc.) y los transforman al formato NGSI para actualizar las entidades correspondientes en un gestor de contexto NGSI. Aunque en estos proyectos se utilizó NGSIv2 y la librería de Python *pyngsi*, desafortunadamente no se actualizó para soportar NGSI-LD, pero la interacción de este componente con el CB es trivial si el desarrollador está familiarizado con este estándar. Además, estos desarrollos contribuyeron a que el doctorando obtuviera la certificación de FIWARE Expert.

La primera decisión consistió en desarrollar un módulo desde cero que fuera simple, fácilmente configurable (*plug-and-play*) e instalable en cualquier nodo del continuo, de manera que estuviera completamente adaptado al Meta-OS sin la necesidad de desplegar componentes adicionales, a parte del CB. Este módulo, desarrollado en Python para reaprovechar estructuras de código existentes, obtiene las características necesarias del nodo de computación de manera automática utilizando librerías como *os, psutil* o *socket*, pero también necesita de la introducción de ciertos parámetros manuales (coordenadas, fuente de energía...). A partir de esta información, junto con los datos de la entidad *Domain* que obtiene del propio CB, crea la entidad correspondiente al nodo en el que se está ejecutando si no ha sido creada previamente.

Consecutivamente, se inicia la ejecución periódica de una función asíncrona (usando la librería *asyncio – Asyncrhonous I/O*) que recolecta las métricas necesarias utilizando las librerías mencionadas con anterioridad para construir un objeto en formato NGSI-LD que se envía como cuerpo de la petición PATCH al endpoint */ngsi-ld/v1/entities/{entityId}* de Orion-LD. De esta manera, la información de contexto relacionada con el nodo de computación queda actualizada.

Además, este módulo expone una API REST ligera (implementada usando *Quart*) incluye un único *endpoint* POST mediante el cual se permite cambiar la frecuencia de ejecución de esta función asíncrona.

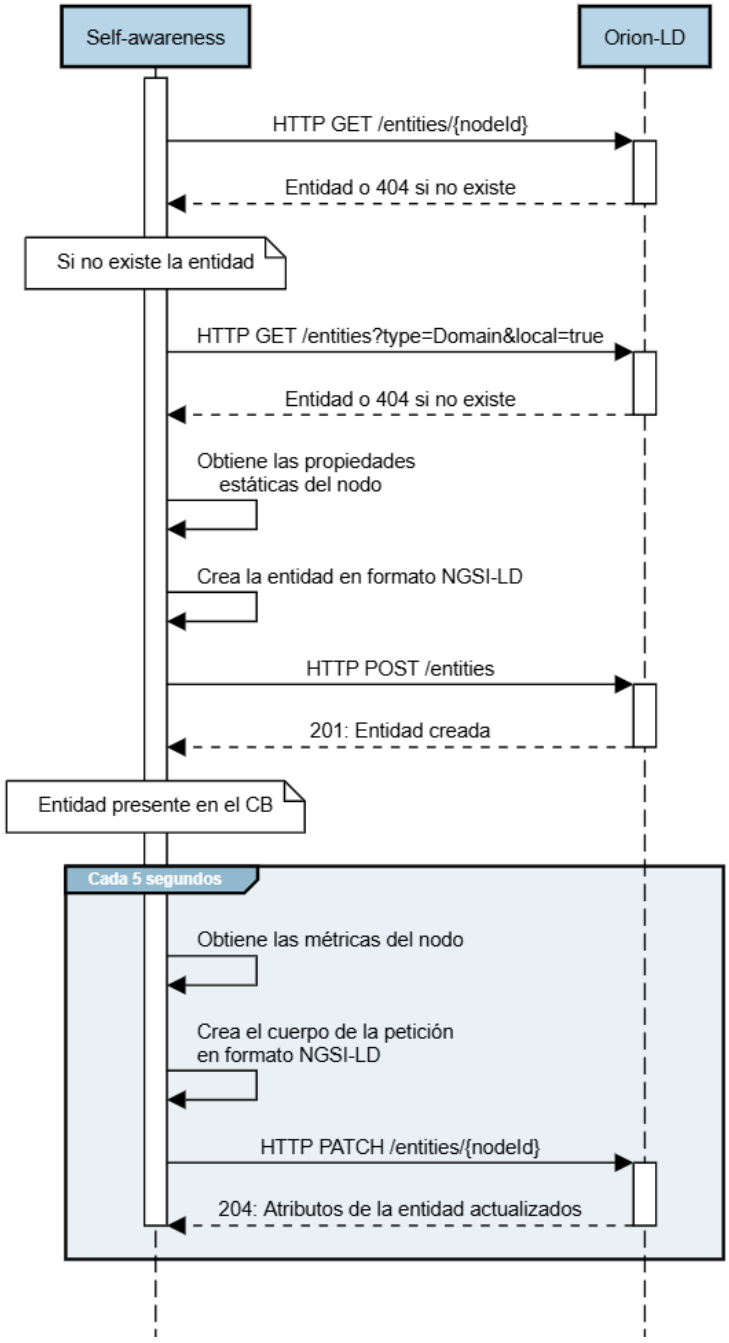


Figura 5.6: Diagrama de secuencia del componente Self-awareness

Node Exporter Adapter

Una de las primeras dudas que surgió durante el desarrollo del elemento de monitorización de los nodos del continuo estuvo relacionada con la necesidad de desarrollar un nuevo elemento de monitorización cuando Prometheus [277] se ha convertido en prácticamente la herramienta de monitorización estándar en despliegues *cloud-native*, en la que el componente *Node Exporter* [278] se encarga de recolectar una colección muy completa que incluye métricas tanto del *hardware* como del OS de las máquinas en las que se instala. Como se ha apuntado anteriormente, en primer lugar se decidió crear un elemento nuevo, pero al comprobar el alto grado de adopción de Prometheus en varios de los entornos a los que se tuvo acceso durante la realización de la tesis (especialmente en clústeres de K8s de proveedores *cloud* en los que se instala de manera nativa), se optó por desarrollar de forma paralela un nuevo elemento de monitorización para aprovechar la potencia de este sistema de monitorización y avanzar hacia un sistema más eficiente, evitando duplicidad de funcionalidades.

La primera tarea consistió en identificar las métricas de interés para el Meta-OS entre el gran número de métricas que recoge el Node Exporter de una máquina, que realmente de tratan de las definidas en el modelo de datos, aunque también sirvió para añadir nuevos atributos a dicho modelo de datos. Por ejemplo, la métrica *node_memory_MemAvailable_bytes* corresponde a la cantidad de memoria disponible en *bytes*, que se debe mapear con el atributo *availableRam* (pero en este caso expresada en MB) de la entidad *ComputingNode*, mientras que otras métricas como la cantidad de memoria utilizada (*currentRamUsage*) o disponible (*availableRam*) se deben calcular manualmente realizando operaciones básicas a partir de diversas métricas para crear la equivalencia correcta. En la siguiente imagen se muestran las métricas originales obtenidas en formato Prometheus y su posterior conversión a formato NGSI-LD para actualizar la entidad correspondiente.

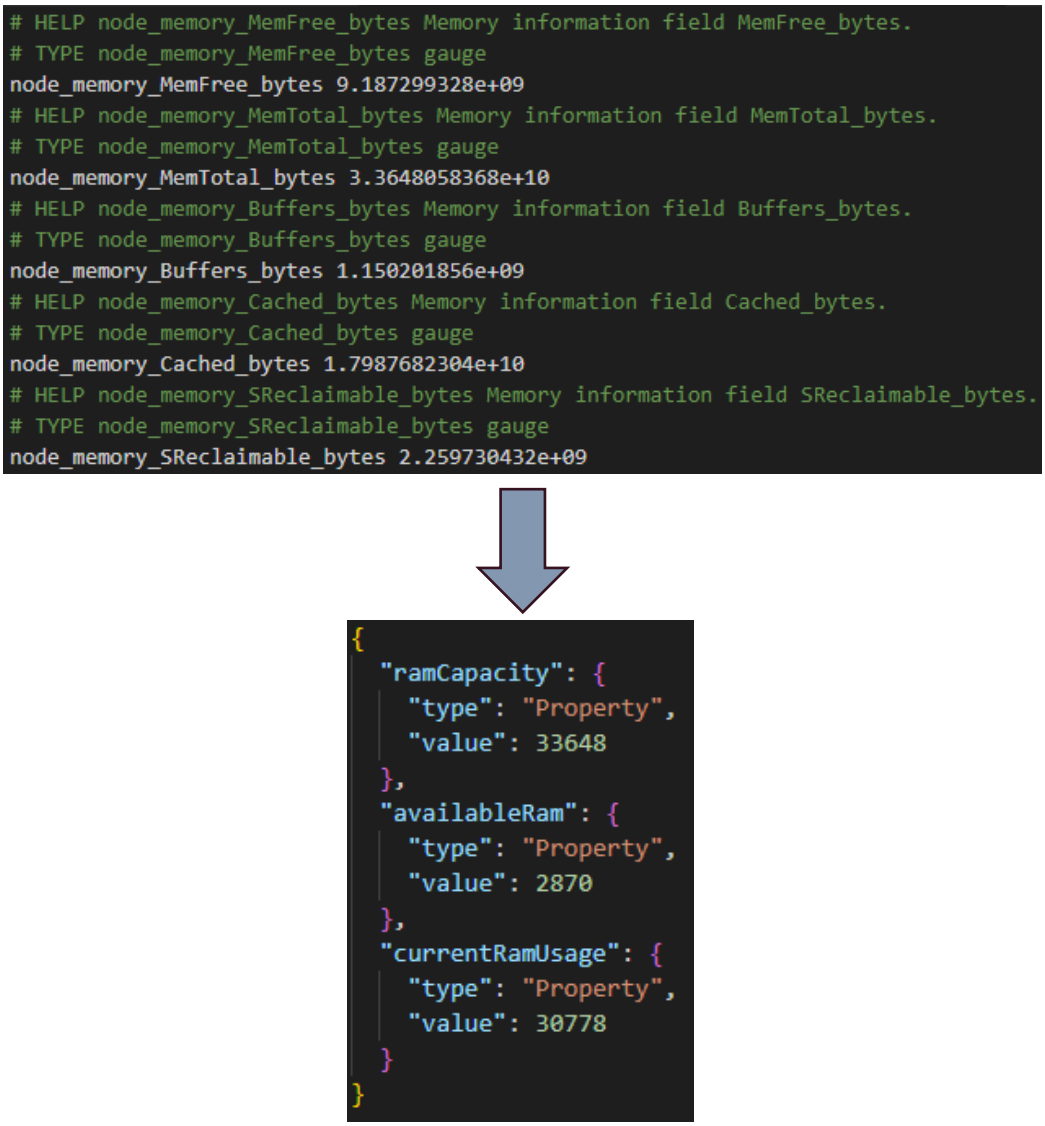


Figura 5.7: Traducción desde formato Prometheus a NGSI-LD para la monitorización de servicios

Go fue elegido como el lenguaje de desarrollo al tratarse del mismo lenguaje en el que han sido desarrollados tanto Prometheus como su Node Exporter, por lo que su librería oficial cuenta con una total compatibilidad con este lenguaje. Además, puesto que ese elemento ha de ser desplegado en todos los nodos del continuo (a no ser que se instale el elemento Self-awareness), debe poder compilarse para el mayor número de arquitecturas de CPU posibles, una característica nativa en Go que facilita la creación de imágenes de contenedor multi arquitectura.

En cuanto a su funcionamiento, excepto en la recolección de la información de contexto sobre el nodo, es prácticamente similar al del anterior módulo, puesto que son componentes equivalentes y reemplazables. Para la recolección de métricas, este elemento realiza una petición HTTP GET al *endpoint /metrics* del Node Exporter para obtener las métricas del nodo en formato texto de Prometheus, al

igual que lo hace el mismo Prometheus, ya que Prometheus funciona como un *scrapper,* es decir, se trata de un elemento activo que recolecta las métricas periódicamente de una serie de *endpoints* en lugar actuar como un agente pasivo al que se le envían las métricas, como actúa Orion-LD en este proceso en concreto. Seguidamente, *parsea* las métricas de interés utilizando la librería del cliente oficial de Prometheus para Go para crear con ellas el cuerpo de la petición HTTP PATCH en formato NGSI-LD, que finalmente se envía a Orion-LD para actualizar la entidad de tipo nodo correspondiente. Este proceso es realizado periódicamente mediante una *goroutine* de manera asíncrona. Además, también se incluye una API REST para cambiar la frecuencia de ejecución de esta *goroutine*.

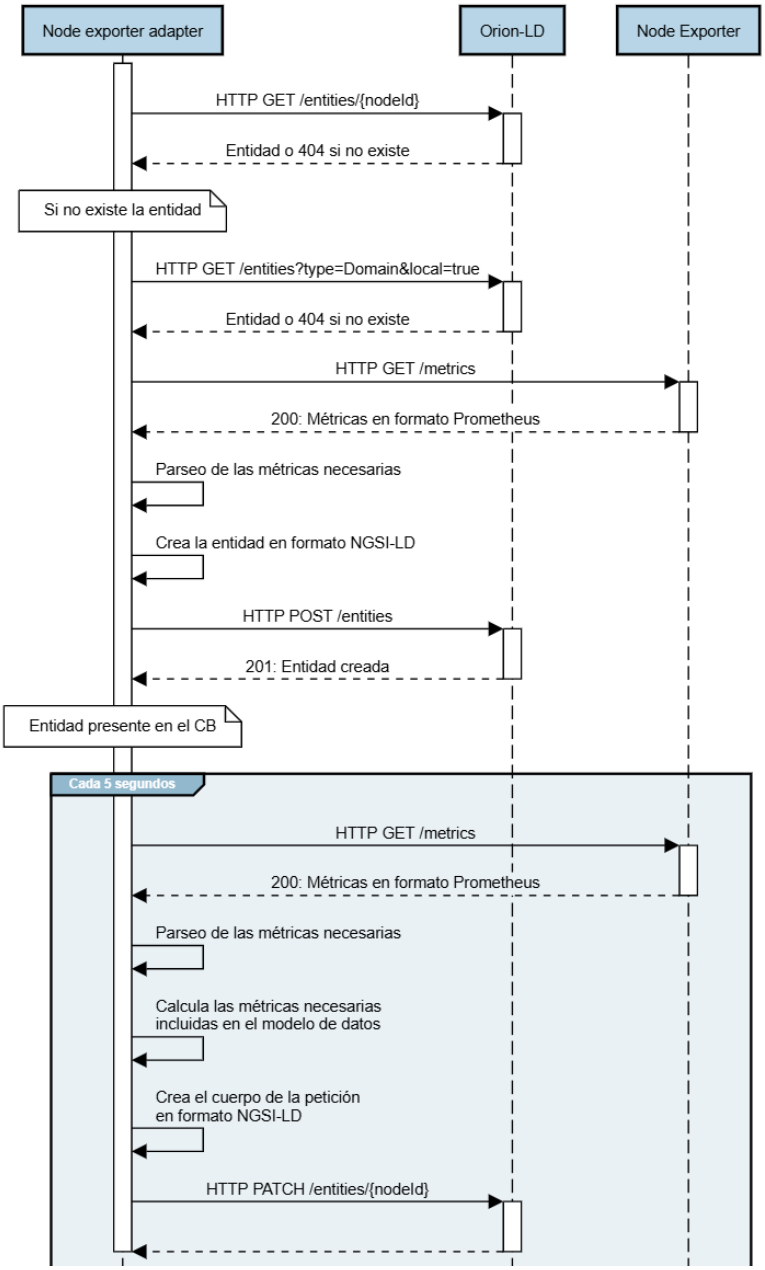


Figura 5.8: Diagrama de secuencia del componente Node Exporter Adapter

5.2.3.2. Monitorización de los servicios

Puesto que la unidad básica de despliegue de servicios del Meta-OS son los contenedores, la decisión más consecuente es usar la herramienta estándar para la monitorización de los contenedores en ejecución: cAdvisor (*Container Advisor*) [279], que a su vez es utilizada por Prometheus para recolectar este tipo de métricas, ya que la expone en su mismo formato. Realmente existe una pequeña diferencia dependiendo del sistema de gestión de contenedores utilizado, puesto que cAdvisor se encuentra integrado por defecto en el elemento *kubelet* de un nodo de K8s. En el caso de otros sistemas, se debería instalar manualmente en cada nodo como requisito previo para el funcionamiento de este módulo. Además, este elemento de monitorización (junto con cAdvisor) también debe instalarse en cada nodo de computación del continuo.

En este momento es necesario destacar que este elemento de monitorización, llamado *Services metrics collector*, no tiene como propósito obtener las métricas relacionadas con los contenedores que forman parte de los componentes que habilitan los bloques básicos del Meta-OS, como por ejemplo los contenedores que corresponden con los módulos que forman HLO en sí, sino que solo se van a tener en cuenta los contenedores correspondientes a servicios desplegados por los usuarios mediante el sistema de orquestación del Meta-OS.

El desarrollo de este elemento también se realizó en Go, por los mismos motivos descritos con anterioridad. En primer lugar, el Services metrics collector obtiene todos los *ServiceComponents* que están desplegados en el nodo en el que se está ejecutando, realizando un filtrado previo. Seguidamente, obtiene todas las métricas de cAdvisor realizando una petición HTTP GET al endpoint */metrics* del mismo. En el caso de K8s, realizaría esta misma petición al endpoint */metrics/cadvisor* del *kubelet*, por lo que se necesita crear un *ServiceAccount* con los debidos permisos en el momento de su despliegue para poder acceder a este endpoint del *kubelet*. Una vez obtenidas estas métricas, utiliza la lista de *ServiceComponents* para filtrar la información necesaria, puesto que cAdvisor no implementa ninguna funcionalidad de filtrado, presentando todas las métricas de todos los contenedores desplegados en el nodo en crudo. Por último, para cada contenedor, con las métricas de interés (*container_memory_usage_bytes* y *container_cpu_load_average_10s*) construye el cuerpo de la petición de actualización NGSI-LD y envía la misma (HTTP PATCH) a Orion-LD para actualizar los atributos *ramUsage* y *cpuLoad* de la entidad de tipo *ServiceComponent* correspondiente.

Por último, también es necesario aclarar que la monitorización del estado de los contenedores correspondientes a los servicios tampoco es responsabilidad de este elemento. De esta tarea se encargan los diferentes LLOs, puesto que son el elemento interactúa directamente con el sistema de gestión de contenedores respectivo y se encargan de gestionar el ciclo de vida completo de los servicios (LCM). Esta funcionalidad se abordará más adelante en la sección 5.4.2.1.

Figura 5.9: Traducción desde formato Prometheus a NGSI-LD para la monitorización de servicios

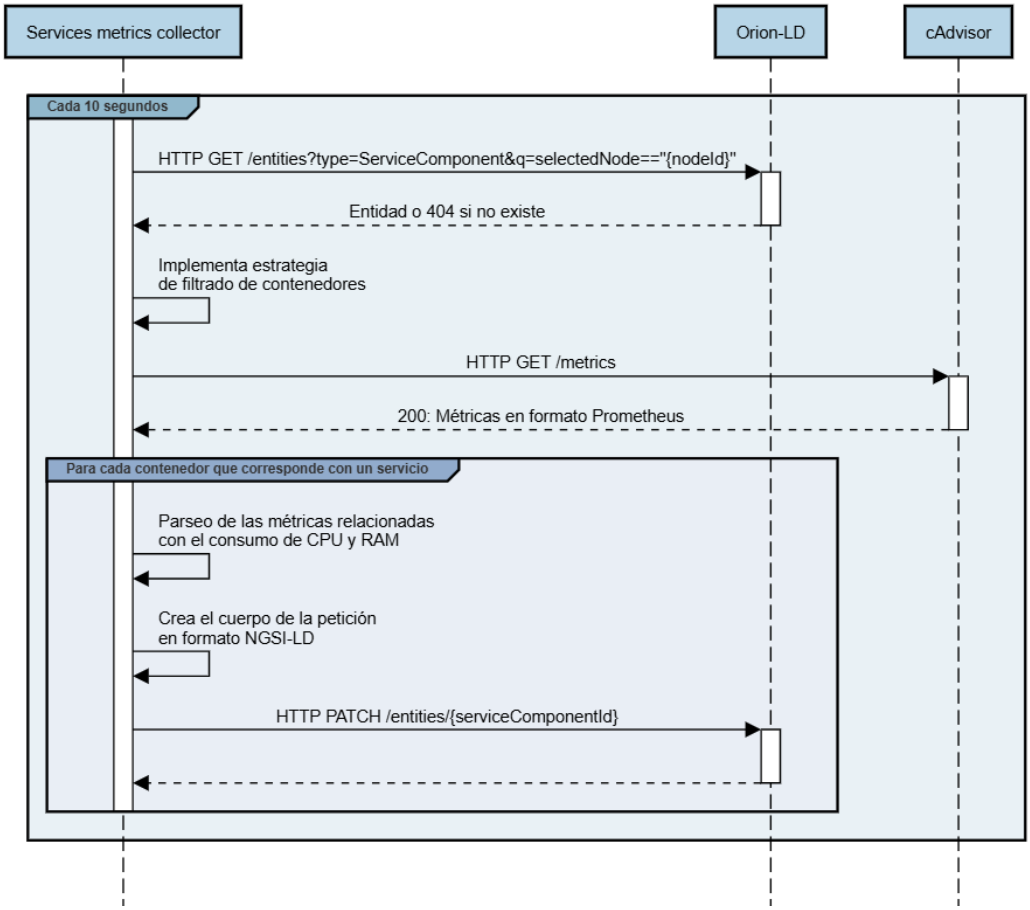


Figura 5.10: Diagrama de secuencia del componente de monitorización de los servicios

5.2.3.3. Monitorización del consumo de energía

La monitorización del consumo de energía, tanto a nivel de nodo de computación como desglosada por contenedor, necesita ser abordada de una manera independiente, puesto que la recolección de este tipo de métricas no es para nada trivial. Además, la obtención de estos valores tiene una mayor dificultad cuando entra en la ecuación infraestructura puramente virtualizada, como las máquinas virtuales que actúan a su vez como nodos del continuo, ya que una multitud de instancias de estas VMs se despliegan sobre la misma infraestructura *hardware* o física. Para mayor dificultad, en estos casos, es necesario disponer de acceso directo a la herramienta de virtualización que se ha utilizado para crear la infraestructura virtualizada sobre el *hardware* (como Proxmox, por ejemplo), para de esta manera poder dotar con los permisos necesarios a las VMs para que puedan acceder a las métricas de consumo reales de la infraestructura *hardware* o física, . Este requerimiento tampoco es, sin lugar a duda, simple, puesto que es necesario descender muy bajo en cuanto a nivel *hardware*.

Existen diversas alternativas para llevar a cabo este tipo de monitorización. Una de las herramientas más básicas es PowerTOP para Linux, sobre la que se construye el elemento *Self-awareness-power-consumption* del proyecto aerOS, cuyo consumo de recursos computaciones es bastante elevado al mismo tiempo que su fiabilidad es reducida, descartándola para nodos con recursos de computación limitados, además de presentar problemas en tipos de *hardware* más específicos.

La herramienta más prometedora para este propósito es Kepler (proyecto tutelado por la CNCF), ya que realiza un análisis completo del consumo de energía (en Watts) estimado de todos los elementos involucrados un clúster de K8s (nodos y contenedores), con un alto nivel de granularidad (permitiendo discriminar a nivel de *namespace, Pod, contenedor* e incluso de proceso del sistema), y ofreciendo estas métricas en formato Prometheus, puesto que realmente se define como un *exporter* de Prometheus. Estas métricas las calcula a partir de los valores de utilización de recursos por parte de los procesos que se están ejecutando, que son recolectados por un programa BPF (Berkeley Packet Filter) integrado en el mismo *kernel* del OS, y de métricas de consumo de energía por parte de la máquina obtenidas en tiempo real de ciertas APIs (Intel RAPL para CPU y DRAM, o NVIDIA NVML para GPU). También permite la instalación de su recolector de métricas directamente en *bare metal* para recolectar ciertas métricas de consumo reales a nivel *hardware* y enviarlas a las VMs directamente. Finalmente, realiza una estimación de las emisiones de CO_2 a partir de estas métricas de consumo de energía [280].

Asimismo, Kepler ha desarrollado y entrenado sus propios modelos de IA para predecir el consumo de energía en el caso de VMs que no tienen acceso a este tipo de métricas, intentando superar una de las mayores limitaciones en cuanto a la recolección de métricas relacionadas con el consumo de energía de la máquina.

Desafortunadamente, la herramienta completa solamente está disponible para ser instalada en K8s, y tampoco puede ser instalada en SBCs como Raspberry Pis.

Finalmente, por todos estos motivos, se ha decidido descartar la monitorización del consumo de energía en el Meta-OS, aunque su inclusión no se descarta en el futuro más inmediato, puesto que el modelo de datos está preparado para su integración, como se refleja en la entidad *PowerSource*. Además, al tratarse de un campo de investigación a la orden del día, que suscita un gran interés y directamente relacionado con los objetivos de sostenibilidad de la Agenda 2030, se prevé que este tipo de herramientas mejoren a corto plazo. Ese tipo de monitorización podría habilitar incluso la certificación de dominios o nodos como respetuosos con el medio ambiente.

5.3.Federación de dominios del continuo

5.3.1. Ubicuidad de la información de contexto

La información de contexto se almacena en tiempo real en el *context broker* NGSI-LD que se despliega en cada dominio de manera independiente, por lo que esta información de contexto con un ámbito local de dominio puede obtenerse realizando las llamadas necesarias a la API de cada CB o utilizando sus mecanismos de publicación y suscripción. Sin embargo, en un contexto como el continuo de computación, es necesario desarrollar un sistema colaborativo con el propósito de hacer accesible esta información —que originalmente tiene un ámbito totalmente local— desde cualquier dominio del continuo, habilitando de esta manera el acceso a la información de contexto de todo el continuo a nivel global.

Este diseño se sustenta en el intercambio de la información de contexto en tiempo real entre los diferentes CB del continuo. De esta manera, como la información de contexto está realmente gestionada por una única instancia de un CB NGSI-LD en cada dominio, se logra evitar la centralización de esta información en único punto, previniendo la creación de un punto único de fallo. Asimismo, se sortea la necesidad de duplicar esta información con la dificultad y baja escalabilidad que conllevaría, puesto que tiene una naturaleza extremadamente dinámica.

A modo de ejemplo, si el CB o un dominio por completo experimenta una caída, lógicamente se perderá el acceso a su información clave de contexto, pero esta caída no supondrá directamente un fallo de funcionamiento global del Meta-OS. Además, esta pérdida de información no afectará a futuros procesos de orquestación, debido a que los nodos de este dominio no estarán disponibles para ser seleccionados.

La ubicuidad de la información de contexto es el principal pilar sobre el que se construye la federación entre dominios del continuo, permitiendo habilitar las

demás funcionalidades colaborativas proporcionadas por el Meta-OS como la orquestación descentralizada, el LCM completo de los servicios desplegados o la plataforma de gestión unificada.

En cuanto a la implementación de este sistema colaborativo, la especificación del estándar NGSI-LD (tal y como se ha descrito en el apartado 2.4.1) incluye mecanismos para la compartición de información de contexto entre varios gestores de contexto y por ende habilitando su uso en entornos distribuidos: las *Context Source Registrations* (CSRs o simplemente *registrations*). Este objeto de configuración específico permite informar a un CB sobre en qué otro CB puede encontrar cierta colección de entidades o incluso de sus atributos [38]. En el caso del Meta-OS desarrollado, la federación de la información de contexto solo se aplica a entidades NGSI-LD por completo, es decir, los atributos de estas entidades no pueden estar distribuidos en diferentes CBs para simplificar este proceso, ya suficientemente complejo por sí mismo, aunque el estándar NGSI-LD contempla esta situación.

La solución propuesta consiste en la creación de una serie predefinida de CSRs (en las que se incluyen los tipos de entidades claves como *ComputingNode* o *ServiceComponent*) de tipo *inclusive* (es decir, un tipo de entidad puede encontrarse en múltiples CBs, a diferencia del tipo *exclusive*) en cada instancia de Orion-LD que apunte a las demás instancias de Orion-LD del continuo (por lo tanto, a cada dominio). De esta manera se establece una relación bidireccional entre todos los gestores de contexto del continuo, habilitando la recolección de la información de contexto demandada (por ejemplo, entidades de un mismo tipo que cumplan con una serie de propiedades) con independencia del CB al que se realice dicha petición.

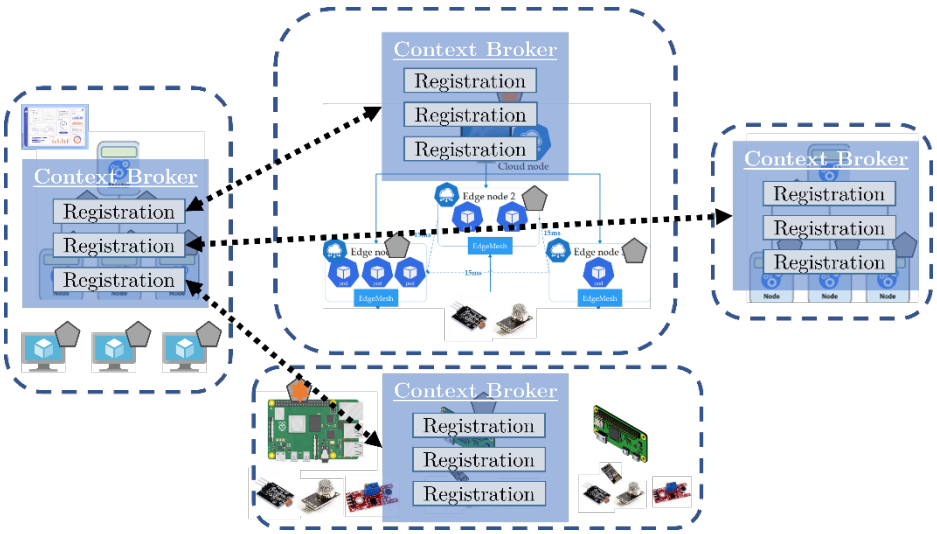


Figura 5.11: Creación de las *Context Source Registrations* en los CBs de los dominios

En la siguiente imagen se muestra un esquema visual en el que se muestra el caso de un continuo compuesto por cuatro dominios. Cada CB de cada dominio cuenta con 3 CSRs (aunque realmente se trataría de una serie de CSRs solo se muestra una por simplicidad) que apuntan a cada uno de los CBs restantes. Por

simplicidad, solamente se muestran los blancos (el *broker* al que apuntan) *de* las CSRs desde el punto de vista de un único CB.

Se han definido tres tipos diferentes de CSRs, una por cada bloque del modelo de datos y con diferentes permisos sobre los datos:

- **Infraestructura:** entidades de los tipos *Domain, ComputingNode* y *LowLevelOrchestrator*. Solamente pueden obtenerse de manera federada (*retrieveOps*), ya que son entidades que reflejan el estado de la infraestructura de un dominio en concreto, por lo que es razonable que solo puedan modificarse en el CB de dicho dominio.
- **Servicios:** entidades de los tipos *Service, ServiceComponent, ComputingNodeRequirements, PersistentStorage, NetworkConnection* y *NetworkPorts*. Pueden obtenerse, modificarse y eliminarse de manera federada (*retrieveOps, updateOps* y *deleteEntity*) para habilitar el proceso de orquestación descentralizado.
- **Usuarios:** entidades de los tipos *User, Role* y *Organization*. Solamente pueden obtenerse de manera federada (*retrieveOps*), al igual que en caso de la infraestructura.


```
{
  "id": "urn:metaos:federation:dominio:infrastructure",
  "type": "ContextSourceRegistration",
  "information": [
    {
      "entities": [
        {
          "type": "Domain"
        },
        {
          "type": "LowLevelOrchestrator"
        },
        {
          "type": "ComputingNode"
        }
      ]
    }
  ],
  "contextSourceInfo": [
    {
      "key": "Authorization",
      "value": "urn:ngsi-ld:request"
    }
  ],
  "mode": "inclusive",
  "operations": [
    "retrieveOps"
  ],
  "hostAlias": "AnotherDomain",
  "management": {
    "localOnly": true
  },
  "endpoint": "https://another-domain.metaos.eu/orionld",
  "metaosDomain": "NCSRD",
  "metaosDomainFederation": true
}
```


Figura 5.12: Ejemplo de una *Context Source Registration* en formato NGSI-LD

En la figura anterior se puede observar el formato de las CSRs que se crean para este propósito. Constan de un identificador único, la lista de tipos de entidades en las que aplica, las cabeceras adicionales que ha de incluir cada petición federada entre CBs (en este ejemplo se reutilizaría la misma cabecera de autorización de la petición original que desencadena la petición federada consecuente), el modo utilizado (siempre será *inclusive* en este Meta-OS), las operaciones permitidas (definidas según el estándar NGSI-LD). Además, es necesario especificar el identificador único (para evitar bucles infinitos) y el *endpoint* a través del cual se expone la API del CB al que apuntan.

En este desarrollo también es obligatorio incluir el campo *localOnly* para evitar que un CB que reciba una petición federada desencadene a su vez múltiples peticiones federadas dando lugar a un bucle infinito, ya que como se ha apuntado previamente, cada CB cuenta con una serie de CSRs que apuntan a los demás CBs del continuo. Por último, los campos *metaosDomain* y *metaosDomainFederation* son adicionales al estándar NGSI-LD, pero se han incluido para identificar inequívocamente estas CSRs esenciales del bloque de federación y permitir un filtrado eficiente de estas CSRs mediante el uso del *query parameter "csf"* (por ejemplo, GET */csourceRegistrations?csf=metaosDomainFederation==true)*, pudiendo descartar otras CSRs que consideren crear los usuarios para otros propósitos.

A medida que se van añadiendo dominios a un continuo, el número de CSRs a crear en cada instancia de Orion-LD aumenta de manera lineal, con una relación de 3 CSRs por cada nuevo dominio. Esto se traduciría, en el caso de un continuo compuesto por 4 dominios, en la creación de 12 *registrations* en cada CB, haciendo un total de 48 *registrations*. Con este ejemplo queda patente la necesidad de desarrollar un componente que se encargue de la gestión de las CSRs en cada dominio, reaccionando en consecuencia a eventos clave relacionados con la adición, suspensión o eliminación de dominios en el continuo. Este elemento se describirá en la siguiente subsección.

En este punto es importante destacar que, aunque se ha elegido Orion-LD como el CB NGSI-LD del Meta-OS (ver 5.2.2), este se podría reemplazar directamente por cualquier otro CB siempre y cuando implemente el estándar NGSI-LD, como por ejemplo NEC Scorpio, hecho que dota de una mayor flexibilidad a este sistema colaborativo de compartición de la información.

Finalmente, antes de adentrarse en los detalles del componente Federador y del repositorio de valores históricos de la información de contexto, se va a detallar las numerosas operaciones internas que suceden entre los diferentes CBs del continuo cuando procesan una petición distribuida, utilizando diagramas de secuencia a modo ilustrativo. En estos diagramas se han excluido las operaciones que Orion-LD realiza sobre su base de datos MongoDB para simplificarlo.

Acceso ubicuo a la información de contexto: entidades y atributos

Para un actor, ya sea un usuario final o un servicio, que pretenda obtener cierta información de contexto global, como por ejemplo todos los nodos de computación del continuo que dispongan de más de 2 GB de memoria RAM, o los servicios desplegados en un nodo en concreto que estén consumiendo una cierta cantidad de memoria, las operaciones internas que realizan los CBs son completamente transparentes. Es decir, realizaría la misma petición GET a la API de cualquier CB NGSI-LD y obtendría la misma respuesta, con el mismo formato.

Internamente, el Orion-LD que recibe la petición original comprueba todas sus CSRs para encontrar coincidencias en base al tipo de entidad y operación demandada, es decir, en el caso de la petición del ejemplo anterior relacionada con los nodos de computación, comprobaría que la CSR contenga permisos de lectura (*retrieveOps*) y que aplicara a este tipo de entidad (type *ComputingNode* incluido en el vector de entidades del campo *information*).

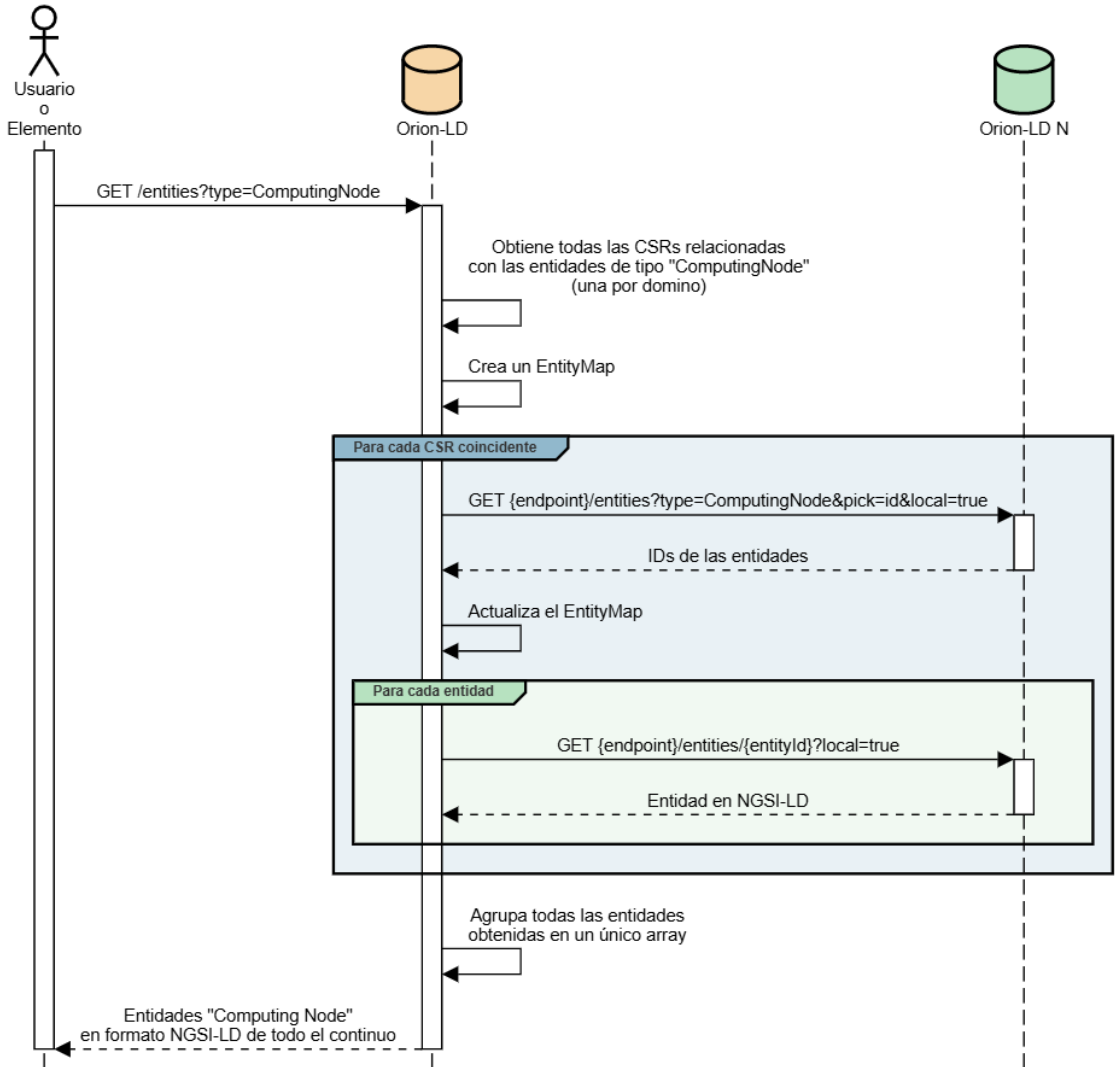


Figura 5.13: Diagrama de secuencia del proceso de obtención de información de contexto de manera federada

Seguidamente, para todas las CSRs coincidentes (en el caso de este Meta-OS una por dominio), el CB envía en primer lugar una petición GET para obtener solamente los ID de las entidades del CB objetivo que cumplan con los requisitos demandados por el usuario (filtrado por tipo de entidad y en base a los parámetros incluidos dentro de la *query parameter* "q"), con las que actualiza un *Entity Map* interno (un *array* que define en qué CB se encuentra cada entidad del tipo buscado, es decir, una relación del ID de la entidad con el ID del CB en el que se encuentra, como se ha explicado en el apartado 2.4.1), que es necesario en el caso de que se necesite hacer uso de la paginación de los resultados. Con esta lista de ID, el CB procede a realizar una petición GET para obtener cada entidad de manera individual, para finalmente agregarlas en la lista final. Finalmente se devuelve un único vector con todas las entidades pertenecientes a todos los CBs de todo el continuo.

Este mismo proceso aplica de la misma manera a las operaciones de actualización de entidades (endpoints PUT, PATCH y POST de la API NGSI-LD) y de eliminación de las mismas (*endpoint* DELETE), en las que se necesitaría incluir otro tipo de permisos en las CSRs, como *updateOps* y *deleteEntity*. Además, NGSI-LD incluye un *query parameter* clave llamado *local*, cuyo valor es booleano, para indicar que las operaciones se ejecuten solamente sobre las entidades pertenecientes al CB al que se envía la petición, es decir, que estén guardadas en su MongoDB local. El uso de este parámetro es interesante para eludir el coste operacional que conllevan las operaciones distribuidas en casos en los que es completamente innecesario, como, por ejemplo, un componente del Meta-OS que desee obtener información sobre el dominio en el que está desplegado (petición GET */entities?type=Domain&local=true*), o el caso importante de los elementos de monitorización descritos en la sección anterior 5.2.3, ya que las entidades que actualizan siempre van a estar en el CB de su propio dominio.

Federación del mecanismo de suscripción

Los CB NGSI-LD cuentan con un mecanismo de publicación y suscripción mediante el cual se pueden crear diferentes suscripciones para que monitoricen los atributos de ciertas entidades y emitan notificaciones como respuesta al cambio de los valores de estos atributos en un formato concreto definido por NGSI-LD. Estas notificaciones pueden emitirse a *endpoints* tipo APIs REST y a *brokers* MQTT, pero en este caso solo se va a contemplar la opción de las API REST.

En el caso de un entorno federado como el de esta tesis, la federación en las suscripciones aplicaría en el caso de que se requiera monitorizar una entidad que se encuentre en un CB diferente al que se ha creado la suscripción, es decir, la suscripción se crea en un dominio, por lo que se espera que el CB de este dominio sea el que emita la notificación de cambios, pero la entidad realmente está almacenada en un domino diferente. Por este motivo, el CB que recibe la petición para crear una nueva suscripción, en primer lugar, comprueba los CSRs que aplican

para las entidades configuradas en esta suscripción. Seguidamente, para todas estas CSRs coincidentes, el CB realiza una petición a los CB objetivos para crear una suscripción subordinada, cuyo *endpoint* de notificación se trata de un *endpoint* singular (*/ex/v1/notify/subscriptions/{subscriptionId}*) de la API del mismo CB emisor. Esta suscripción subordinada es realmente la que monitoriza la entidad local, para emitir (como si se tratara de una suscripción al uso) una notificación al CB "padre" cuando cambian los valores de los atributos configurados. El doctorando puede afirmar que este mecanismo distribuido no es para nada trivial, puesto que implica una gran coordinación entre todos los CBs implicados, como ha podido comprobar al haber estado involucrado en el proceso de desarrollo y *testing* de la versión preliminar de esta funcionalidad en Orion-LD.

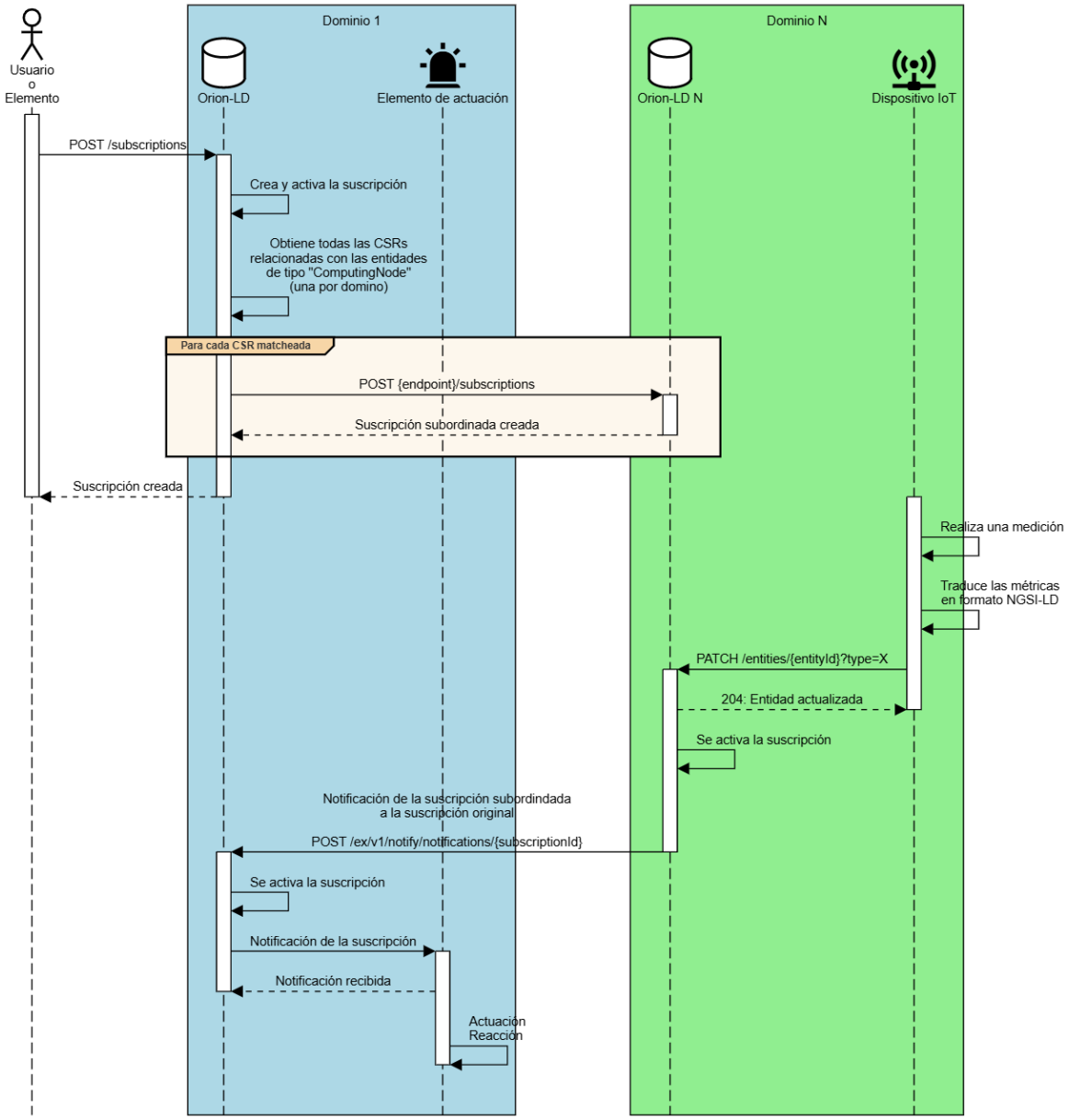


Figura 5.14: Caso de uso del mecanismo de suscripciones federadas de Orion-LD

El caso de uso más directo estaría relacionado con el despliegue en un dominio de un elemento que requiera monitorizar los *ServiceComponents* (u otras entidades del modelo de datos) de todo el continuo o una única instancia con independencia del dominio en el que se encuentre. Por lo tanto, crearía una suscripción en el Orion-LD de su propio dominio, pero esperaría recibir notificaciones relacionadas con las entidades de todo el continuo.

Otro caso de uso posible más relacionado con el ámbito IoT consistiría en la creación de una suscripción por parte de una aplicación en un dominio para recibir actualizaciones sobre las mediciones realizadas por un dispositivo IoT que se encuentra desplegado en otro dominio. De esta manera esta aplicación tendría acceso a estos datos IoT de monitorización, con independencia del dominio en el que esté desplegada, incluso podría actuar sobre ella. Este caso de uso es el que se representa en el diagrama de secuencia de la Figura 5.14.

5.3.1.1. Componente Federador

La elevada cantidad de CSRs que se necesitan gestionar en cada instancia del CB NGSI-LD evidencia la necesidad de un componente que automatice dicho proceso, reaccionando en consecuencia a eventos clave relacionados con la adición, suspensión o eliminación de dominios en el continuo. Este elemento, llamado Federador, cuenta con una arquitectura distribuida, de manera que se despliega una única instancia en cada dominio para realizar operaciones exclusivamente sobre la instancia local del CB. Este componente ha sido diseñado para realizar procesos colaborativos en los que intervienen las demás instancias del mismo pertenecientes a los demás dominios, siendo en cierta manera equivalente a las operaciones distribuidas de los CB NGSI-LD.

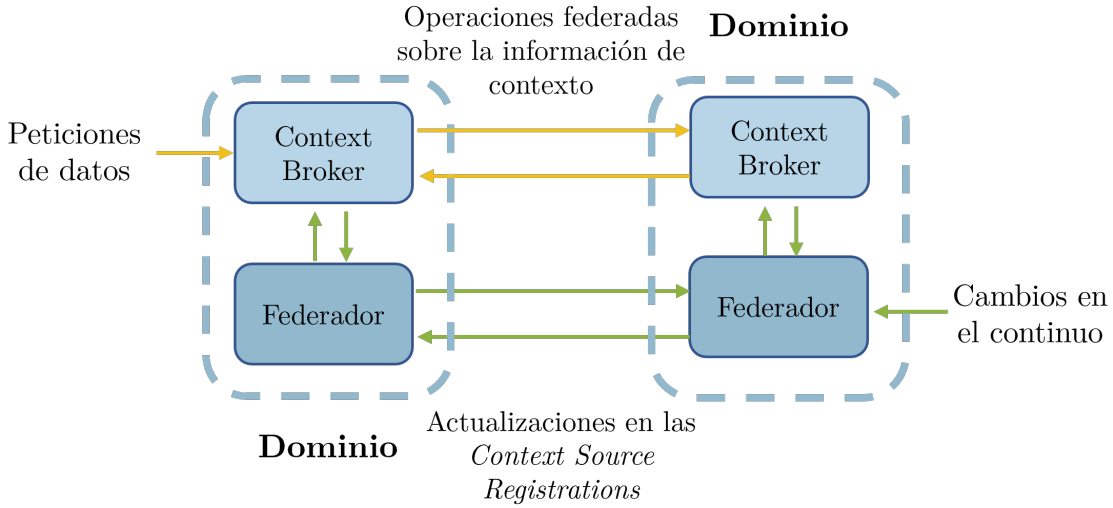


Figura 5.15: Diagrama de bloques del componente Federador en el continuo

Cada instancia del componente Federador debe tener asignada un Federador de cualquier otro dominio (llamado *peer federator*) para poder realizar el proceso de unión al continuo, y que este pueda, a su vez, diseminar la creación del nuevo dominio, además de proveer al nuevo Federador con el registro actual de dominios del continuo. Además, puesto que la información de contexto (y, por lo tanto, del registro de dominios) se encuentra distribuida entre los CBs de cada dominio, este componente cuenta con una pequeña base de datos auxiliar del tipo *key-value* para almacenar información importante, como el listado completo de dominios, que podría ser útil en caso del fallo de un dominio.

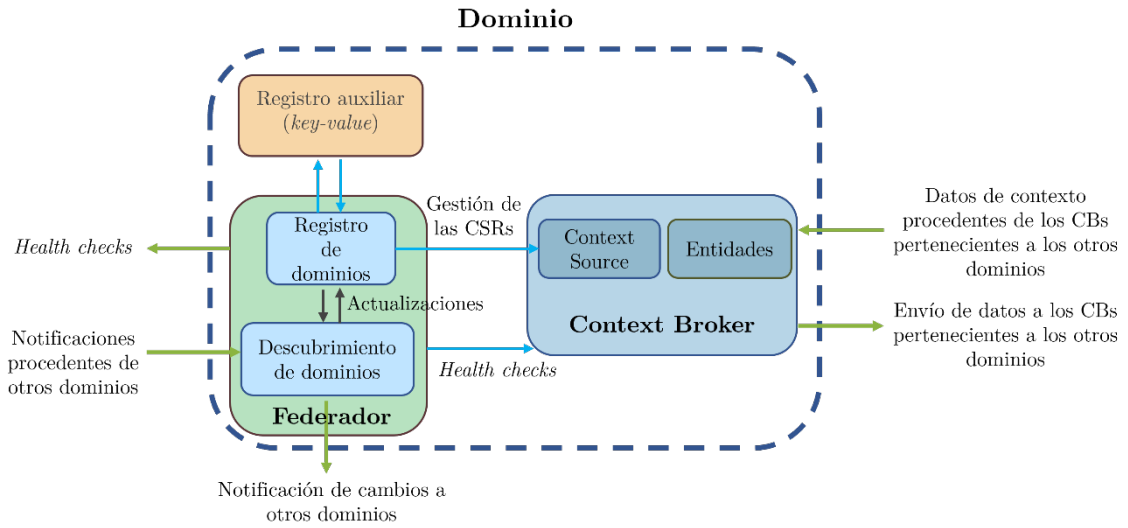


Figura 5.16: Diagrama de bloques del componente Federador en un dominio

Hasta este momento se ha referido a la información de contexto solamente teniendo en cuenta las entidades relacionadas con el modelo de datos del continuo. Sin embargo, la ubicuidad en el acceso a esta información permite habilitar un sistema de compartición de datos clave entre servicios desplegados en el continuo, como si se tratara de un *broker* o canal de mensajería, a través de los CBs. Por este motivo se ha introducido en la API REST de este componente un *endpoint* adicional para federar otro tipo de entidades que no esté completada en los tres tipos de CSRs descritos en esta sección.

En cuanto a su desarrollo, este componente ha sido desarrollado utilizando el lenguaje Go e implementa una API como elemento central, que incluye endpoints para la creación, eliminación, habilitación en inhabilitación de dominios en el continuo. En un primer momento se pretendió desarrollar esta API utilizando gRPC en lugar de REST, pero se descartó finalmente debido a que este elemento ha de interactuar con Orion-LD utilizando su API REST, por lo que las mejoras que se obtendrían serían en cierta parte limitadas.

A modo ilustrativo, en el siguiente diagrama de secuencia se muestra la interacción entre los distintos Federadores del continuo, junto los procesos internos

y su interacción con las instancias de Orion-LD de su dominio, para el caso en el que se añade un nuevo dominio al continuo de computación.

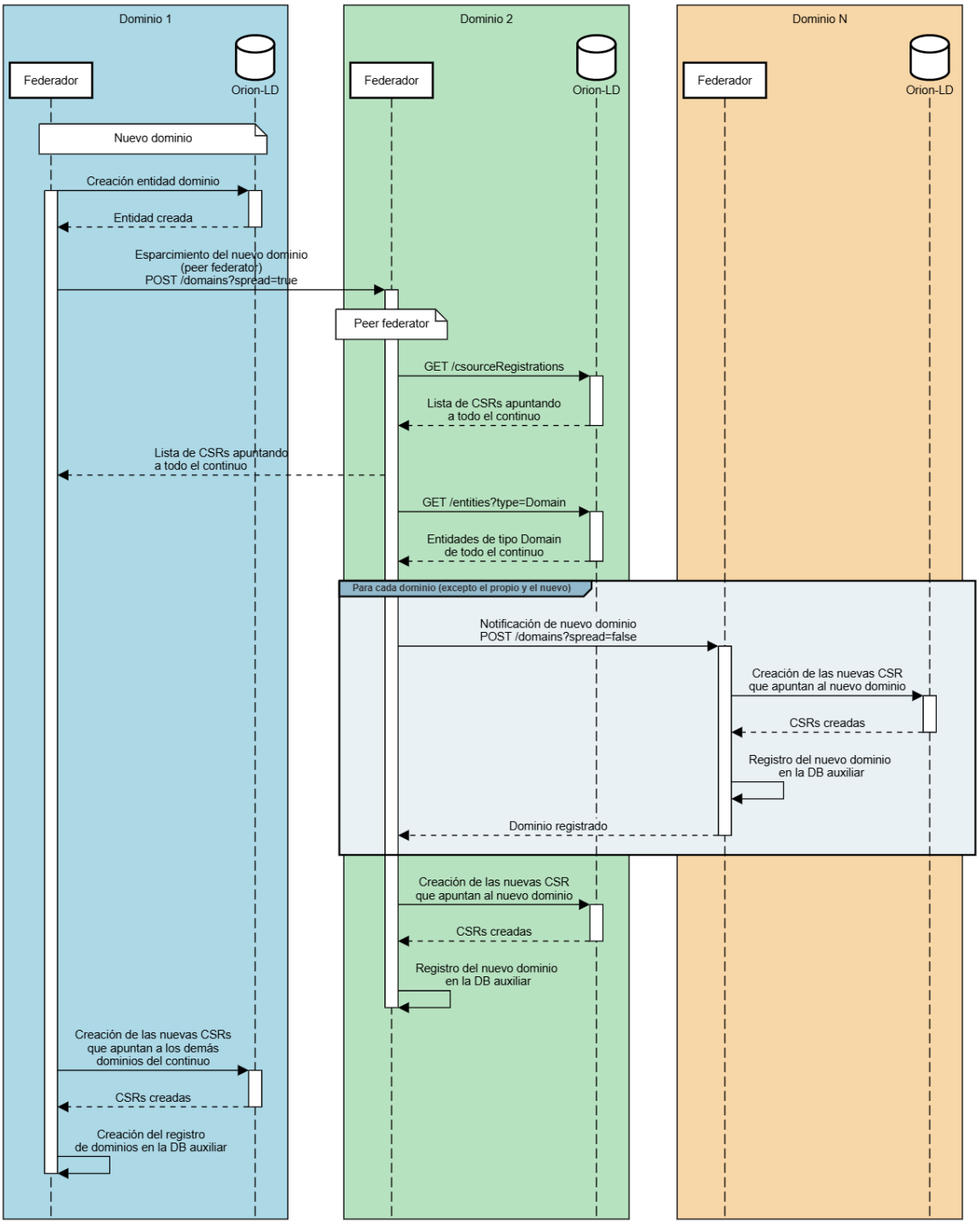


Figura 5.17: Diagrama de secuencia de la adición de un nuevo dominio en el continuo

5.3.1.2. Repositorio de datos históricos

En el apartado 5.2.2 se han descrito algunas de las numerosas opciones disponibles para persistir los valores históricos de los atributos de las entidades gestionadas por Orion-LD. Desafortunadamente, estas alternativas solamente aplican para las entidades locales que están almacenadas realmente en el MongoDB de cada instancia de Orion-LD, por lo que, en el caso del continuo, solamente permitirían crear un repositorio de datos históricos locales de cada dominio, sin establecer una conexión entre ellas o un método de agregación común de esta información. Además, habría que añadir los costes computacionales que conllevaría desplegar una instancia de la solución elegida (Mintaka, por ejemplo), junto con la base de datos asociada en la que realmente se almacenan los datos, en cada uno de los dominios.

El doctorando ha desarrollado una solución para la creación de un repositorio de datos históricos general para todo el continuo, incluyendo las entidades requeridas de todos los CBs. Lamentablemente, también está lejos de ser una opción perfecta, puesto que se aleja un poco del paradigma de computación distribuida sobre el que se sustenta este Meta-OS. Esto es debido a que esta solución implementa un repositorio único y centralizado, de manera que los datos históricos no están distribuidos o federados. Por otra parte, es necesario resaltar que el proceso de recolección de estos datos en tiempo real sí que se realiza de manera distribuida, ya que se construye tomando como base el mecanismo de suscripciones distribuidas descrito previamente.

Esta solución consiste en desplegar una instancia adicional de Orion-LD (junto con su base de datos MongoDB [274]) en un dominio (la elección lógica sería el *entrypoint*) con el parámetro *TRoE* (*Temporal Representation of Entitites*) activado, para que guarde en la base de datos TimescaleDB [281] un registro del valor histórico de los atributos de todas sus entidades almacenadas. En esta instancia de Orion-LD no se van a crear entidades manualmente, sino que se crean a partir de una CSR que especifique los tipos de las entidades que se quieran persistir (presumiblemente las relacionadas con el modelo de datos del continuo) apuntando a cada CB del continuo, incluyendo el del propio dominio. Seguidamente, se crea una única suscripción para que monitorice todos (o únicamente los que sean de interés) los atributos de las entidades incluidas en las CSRs, con un *endpoint* de notificación especial de Orion-LD: */ex/v1/notify*. De esta manera, cada vez que se actualice cualquier de estas entidades en cualquier Orion-LD del continuo, la notificación se recibirá en este Orion-LD "especial" o de persistencia, y persistirá los nuevos valores en la base de datos histórica TimescaleDB. Los datos de esta base de datos pueden ser usados como requiera el usuario, siendo una opción el despliegue del componente Mintaka de FIWARE, el cual se trata de una API enfocada a la realización de búsquedas y agregación temporal de la información [282].

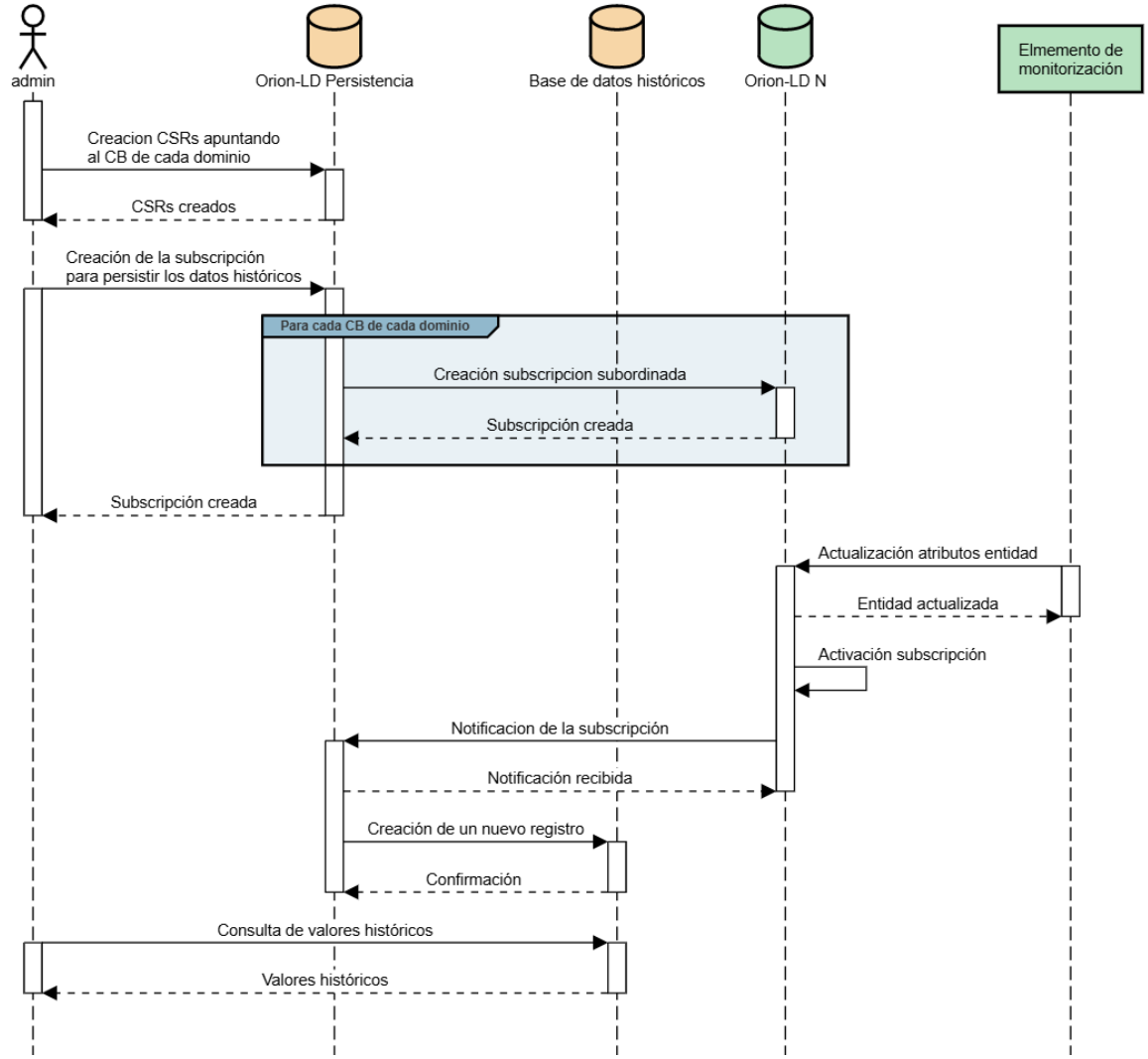


Figura 5.18: Diagrama de secuencia del repositorio de datos históricos

5.3.2. Comunicaciones entre dominios

La federación entre los dominios que conforman el continuo de computación *Cloud-Edge-IoT* es uno de los bloques esenciales que habilitan la creación del Meta-OS desarrollado durante esta tesis doctoral, cuyas características se han descrito en el apartado 4.5.1. De acuerdo con estas características, es necesaria la creación de un sistema para habilitar las comunicaciones necesarias entre los dominios, entre las que se incluyen las peticiones federadas entre los diferentes CBs para el acceso ubicuo a la información de contexto, descrito en el subapartado anterior, o el intercambio de información entre los HLOs del bloque de orquestación de servicios, que se describirá durante la próxima sección 5.4.3. Asimismo, es necesario considerar que los dominios pueden estar dispersos geográfica o administrativamente, así como estar conectados a diferentes tipos de redes.

La solución más directa consistiría en definir un requisito estricto en el diseño de los dominios, estableciendo que cada uno disponga de una dirección IP pública. Esta IP pública debería estar expuesta a la red global, o al menos, poder ser enrutada por los demás dominios del continuo en el caso de continuos con un ámbito privado o local, como, por ejemplo, una empresa o una universidad en la que los dominios y sus nodos de computación estén conectados a su red corporativa. Sin embargo, esta solución dista de ser plenamente aplicable, puesto que solo encajaría en casos de uso ideales en los que se disponga de un alto grado de control sobre la infraestructura, o que su arquitectura esté más bien cerca de ser *cloud-native*. Este hecho apartaría a esta solución, y por tanto al Meta-OS, de entornos de red más complejos, que son los que suelen presentarse en entornos relacionados con el *edge computing* y el continuo de computación en sí.

Descartado este primer diseño, la acción más lógica consiste en basar la arquitectura de la solución en una de las herramientas para la conexión de elementos en entornos altamente heterogéneos estudiadas en la sección 2.5. En este ámbito destaca la herramienta *libp2p*, dado que permite la creación de redes descentralizadas y fácilmente adaptables a las características de diferentes entornos de red [70]. Además, está basada en la capa de aplicación del modelo OSI e implementa sus propios mecanismos de NAT transversal. Por estos motivos, el doctorando decidió utilizar esta herramienta para crear una red P2P entre los diferentes dominios que conforman el continuo.

Debido a que los dominios son realmente entidades virtuales o administrativas, la red P2P ha de establecerse entre nodos pertenecientes a cada dominio, es decir, un nodo seleccionado de cada dominio se unirá a la red como *peer* de la misma. Este nodo seleccionado (bajo el criterio del administrador del dominio) se tratará potencialmente del nodo que está ejecutando el CB o que tenga algún tipo de acceso a una IP pública, ya sea directamente o mediante un *proxy* intermedio, por ejemplo. Para habilitar esta red, en cada uno de estos nodos se tiene que desplegar un agente (llamado agente *libp2p*) que ha sido desarrollado utilizando Go (al igual que la misma herramienta *libp2p*, y también debido a que la implementación de la librería de *libp2p* para este lenguaje es completa) [44].

El proceso para la configuración y el establecimiento de esta red P2P se ve claramente beneficiado gracias a que se conoce de antemano (desde el momento en el que se diseñan los dominios antes del despliegue del Meta-OS, como se apuntó en la sección 4.2) la naturaleza de los dominios, es decir, si se tratan de dominios público o privados. Además, esta información se persiste en el CB del dominio (entidad *Domain*), así como su *PeerID* o su *multiaddress* asociadas en la red P2P, acción realizada por el agente cuando se inicia.

Cuando se añade un domino privado al continuo, el Federador de éste contacta directamente con su homólogo de un dominio público (ya que es directamente accesible sin necesidad de la red P2P) para obtener la lista de dominios, y, por lo tanto, la lista de *peers* de la red P2P. De esta manera, gracias a la ubicuidad en el acceso a la información de contexto proporcionada por el Meta-

OS se logra evitar la ejecución de los procesos de descubrimiento de *peers* que utiliza *libp2p* (Kad-DHT, mDNS...), con el consecuente ahorro de los recursos de computación y la elusión de posibles errores (*peers* no encontrados, etc.).

Con esta lista de *peers*, para cada dominio privado, el agente *libp2p* del dominio establece una conexión con el nodo *relay* seleccionado, para después intentar establecer una conexión directa con el nodo del dominio privado utilizando el protocolo DCUtR (Direct Connection Upgrade through Relay), que utiliza técnicas de *hole punching* [76]. Si esta conexión directa no es posible, se establece un mecanismo alternativo en el que el tráfico entre ambos dominios privados (realmente entre sus nodos que actúan como *peers*) se envía a través del dominio público (su nodo *relay*). Sin embargo, esta alternativa conlleva asociada un aumento del tráfico soportado por el nodo *relay* y un aumento de la latencia en las comunicaciones [44].

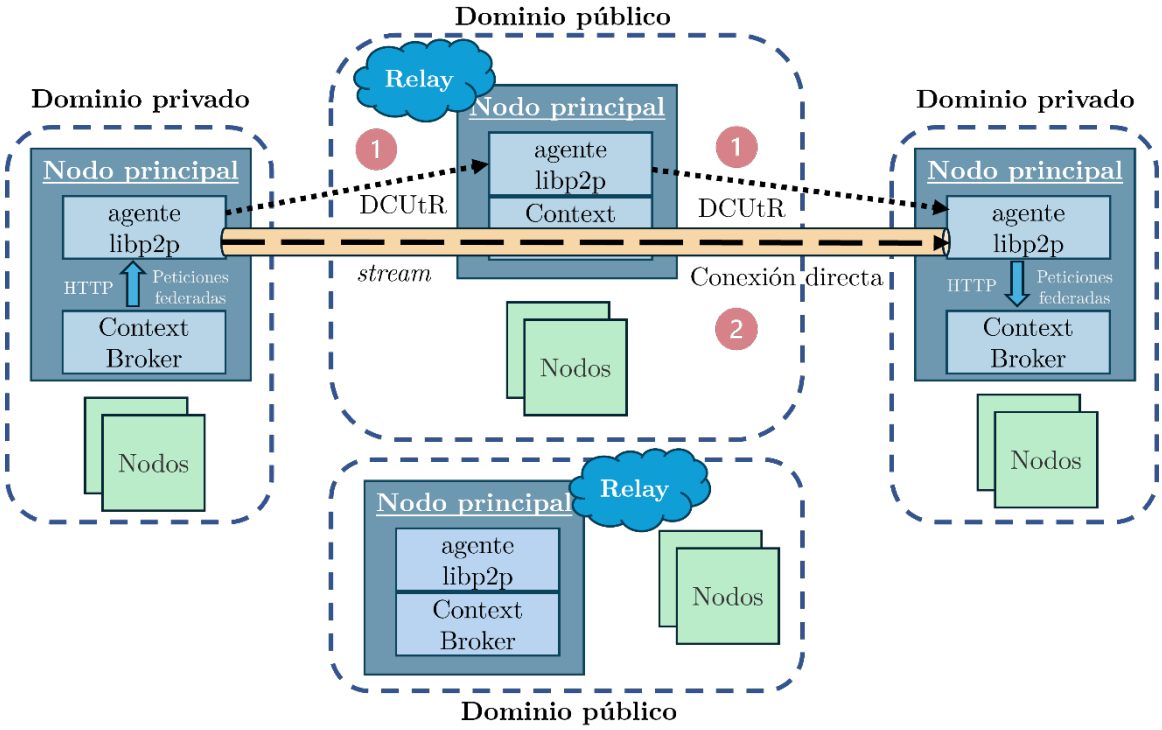


Figura 5.19: Conexión entre dominios privados a través de la red P2P

En cuanto a las comunicaciones entre los elementos de los bloques básicos del Meta-OS, estas deben realizarse a través de los agentes *libp2p* de cada dominio. Estos agentes crean un *stream* (un canal de comunicaciones bidireccional) entre ellos para cada uno de los elementos que se han de comunicar entre dominios (HLO, CB, Federador...), puesto que *libp2p* permite la creación de diferentes conexiones virtuales (*streams*) sobre una misma conexión real entre *peers* (la conexión previamente creada entre cada agente) [283]. De esta manera se evita que se tengan que establecer nuevas conexiones para cada elemento, ya que el consumo de recursos de este proceso es alto (sobre todo para las conexiones entre nodos privados).

Debido a que este proceso debe ser transparente a los elementos que se comunican entre dominios (no han de ser conscientes de la existencia de la red P2P), es necesaria la implementación de un *proxy* transparente en el agente a modo de intermediario. Con ello, estos elementos (HLO, CB, Federador...) realizan peticiones HTTP a este *proxy* pensando que en realidad están realizándolas a sus homólogos de los demás dominios, sin percatarse de su existencia. El *proxy* se encarga de introducir en el *stream* de *libp2p* correspondiente (por ejemplo, */hlo/1.0.0*) la petición en cuestión para que sea recibida por el agente *libp2p* del dominio receptor, que finalmente entrega la petición al elemento destinatario (HLO, CB, Federador...). Seguidamente, la respuesta a la petición es recibida por el agente, que la introduce en el *stream* para que sea recibida por el agente del dominio emisor, y finalmente entregada al elemento que ha emitido la petición original (HLO, CB, Federador...), siguiendo el orden inverso.

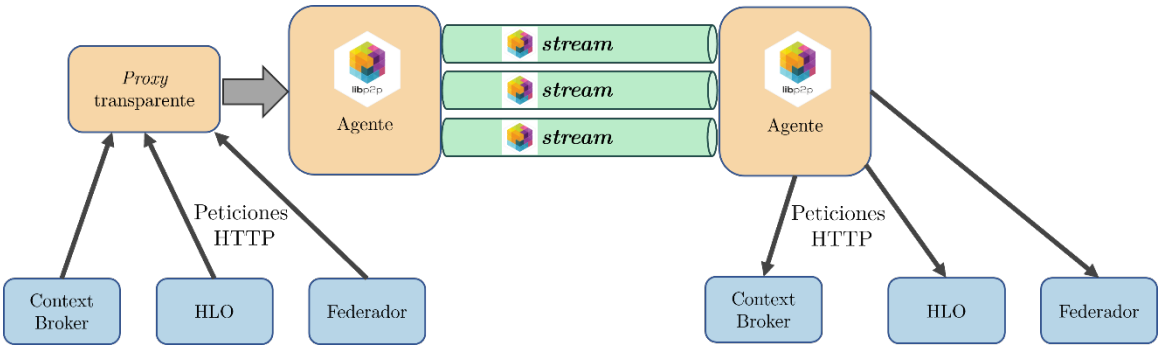


Figura 5.20: Comunicaciones entre elementos a través de la red P2P

5.4. Orquestación y despliegue de servicios

En la sección 4.5.2 se describieron los bloques generales que forman parte del proceso de orquestación descentralizado de servicios del Meta-OS, que sirvieron de inspiración para la definición de la arquitectura de orquestación para el continuo de computación que pretende adoptar la iniciativa EUCEI (introducida en el apartado 3.4). Esta arquitectura, recordemos, está formada por dos bloques jerárquicos principales (HLO y LLO), que se sustentan sobre la federación entre los dominios que forman el continuo. En esta sección se pretende profundizar en la implementación de esta arquitectura, realizando una descripción detallada del funcionamiento de este proceso de orquestación, así como de los elementos y procesos que lo conforman a nivel tecnológico.

Antes de adentrarse en los aspectos técnicos, es conveniente realizar unas aclaraciones de las que ya se habían ido dando pequeñas pinceladas hasta el momento. Los usuarios del Meta-OS especifican el despliegue de un servicio completo, ya sea a través del portal de gestión (próxima sección 5.5) o directamente a través de un HLO, que está compuesto por diferentes componentes que corresponden con los contenedores que se desplegarán en los nodos, como se refleja en modelo de datos diseñado que se mostró en la sección 5.2.1. Aunque el HLO reciba la petición del servicio completo (*Service*), a partir de este momento, el

proceso de orquestación se realiza de manera independiente para cada componente (*ServiceComponents*), pero siempre asegurándose de dotar de una cierta coherencia a los mismos en cuanto al servicio al que pertenecen.

El siguiente diagrama de secuencia pretende servir como introducción al contenido de esta sección, puesto que muestra el proceso completo de orquestación descentralizado de un servicio, incluyendo las interacciones entre los componentes internos del HLO, LLO y el gestor de contexto que serán introducidos en las próximas subsecciones. Se ha elegido el proceso de despliegue de un servicio de manera automática debido a que se trata del proceso más complejo y representativo de los contemplados por el sistema de orquestación del Meta-OS, y que, además, ya se había descrito en rasgos generales en el apartado 4.5.2. Este hecho no pretende restar importancia a los demás procesos manejados por el mismo (actualización, eliminación o reorquestación de un servicio; despliegue manual o semiautomático de un servicio, etc.), que también se describirán posteriormente.

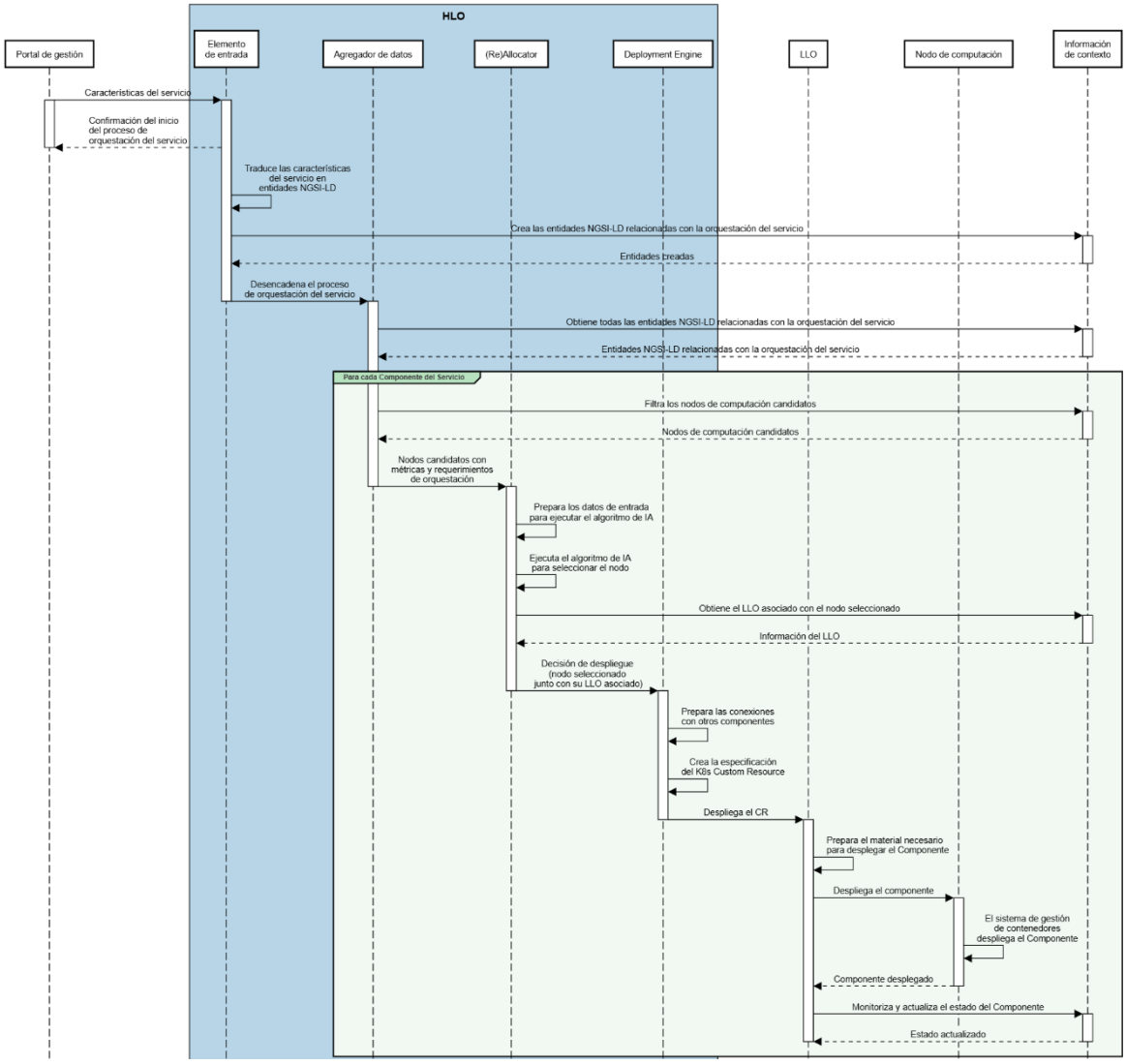


Figura 5.21: Diagrama de secuencia del proceso de orquestación de un servicio

5.4.1. Entrypoint balancer

El elemento Entrypoint balancer, así como su concepto, ha sido diseñado para evitar la creación de un punto adicional de centralización en el proceso de orquestación del Meta-OS, que por definición es completamente descentralizado. Su propósito es simple y conciso: distribuir las diversas peticiones de orquestación de servicios emitidas por los usuarios entre los diferentes dominios que componen el continuo de computación, ya que, de no existir, todas las peticiones serían procesadas por el HLO del dominio *entrypoint*, introduciendo un punto de centralización así como un cuello de botella para todo el proceso. La creación de un elemento como el Entrypoint balancer es posible gracias a la misma federación del continuo, puesto que todos los HLOs tienen acceso a la información de contexto clave de todo el continuo gracias a la ubicuidad en el acceso a esta información proporcionada por el Meta-OS (ver sección 5.3.1). Además, como indica su propio nombre, está ampliamente basado en los balanceadores de carga (*load balancers*) de red que son extensivamente utilizados en entornos del *cloud computing*.

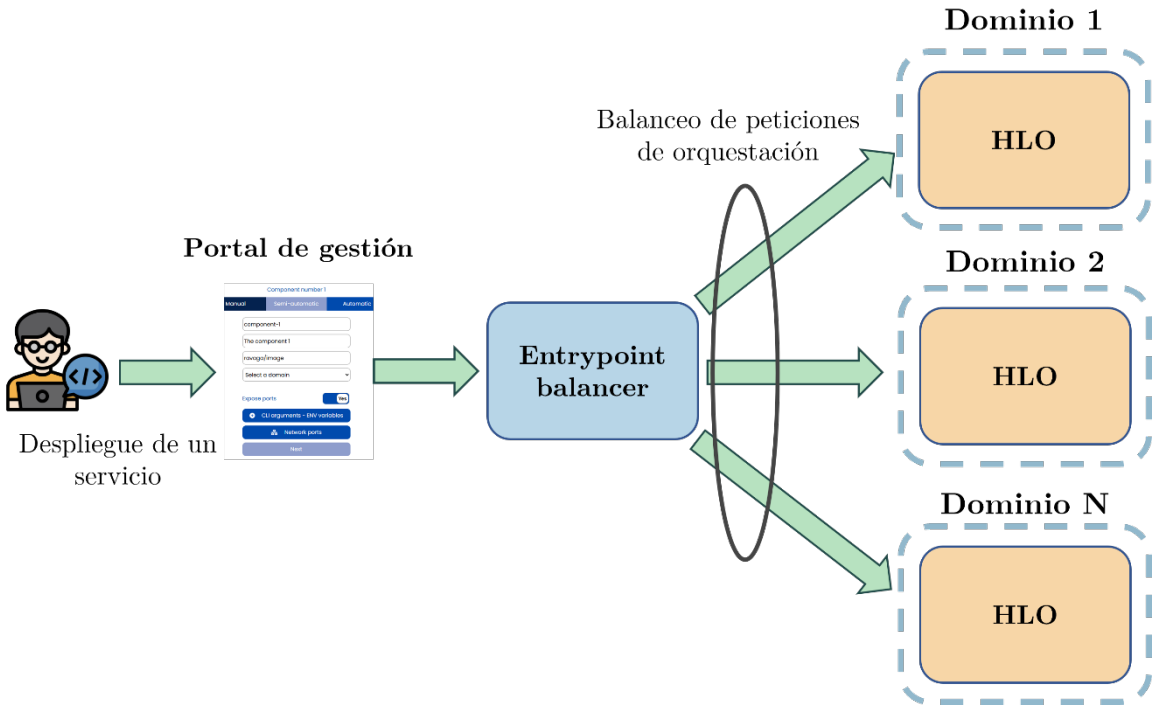


Figura 5.22: Entrypoint balancer en el proceso de orquestación

Los algoritmos empleados para la implementación de balanceadores de carga de red se sustentan sobre una gran base teórica, constituyendo un campo de investigación propio, que está fuera del alcance de esta tesis doctoral. En el contexto de esta tesis, la innovación aportada por el Entrypoint balancer está más relacionada con el concepto en sí mismo, puesto que la decisión tomada por este no tiene una gran influencia en el proceso de orquestación ni en su resultado. En consecuencia, se ha optado por realizar una implementación básica del Entrypoint

balancer con el objetivo de validar el concepto, basada en el uso e implementación adaptada de un balanceador de red del tipo *Weighted Least Connections*, que consiste en el envío de la petición al elemento receptor que cuenta con menos conexiones activas mediante la asignación de pesos [284]. Concretamente, se ha desarrollado utilizando Java con Spring Framework para mantener una cierta coherencia con el elemento Backend de la plataforma de gestión unificada.

En primer lugar, el Entrypoint balancer obtiene todas las entidades NGSI-LD del tipo *Domain* del continuo (gracias a la ubicuidad en el acceso a esta información) y marca todos los dominios como disponibles, lo cual significa que todos van a ser considerados para la próxima iteración del algoritmo. Seguidamente, para cada dominio se obtiene la información de sus nodos de computación, que utilizan como parámetros de entrada para ejecutar la función de *weighting* (o asignación de pesos), con el objetivo de calcular el peso de cada dominio utilizando la fórmula especificada. La puntuación final de cada dominio se obtiene dividiendo el número de componentes desplegados en sus nodos (en estado diferente a *Finished*), entre el peso de cada dominio. Finalmente, la petición de orquestación se envía al HLO del dominio con mayor puntuación. El desarrollo de este módulo ha sido realizado para que la función de asignación de pesos, así como la del cálculo de la puntuación final, sean fácilmente reemplazables, concretamente mediante el uso de interfaces de Java. De esta manera, se logran fijar los argumentos de entrada (entidades NGSI-LD de los nodos y los servicios) y de salida (resultado con formato *Float*) de estas funciones, pero dejando total libertad a los desarrolladores para implementar la lógica de cálculo.

$$Peso\ de\ un\ dominio = \frac{\sum \overline{(uso\ de\ CPU\ y\ de\ RAM)\ de\ cada\ nodo}}{Número\ de\ nodos\ del\ dominio} \quad (1)$$

$$Puntuación\ de\ un\ dominio = \frac{Número\ de\ componentes\ desplegados}{Peso\ del\ dominio} \quad (2)$$

Para evitar la sobrecarga de dominios específicos, se ha introducido una variable global denominada *maxAssignments*, que permite configurar el número máximo de asignaciones consecutivas que un dominio puede recibir. Si un dominio supera este umbral, entra en cuarentena, lo que significa que será excluido temporalmente del balanceo durante un total de *maxAssignments - 1* iteraciones.

5.4.2. Aspectos transversales

5.4.2.1. Gestión del ciclo de vida de los servicios desplegados

Uno de los requisitos definidos en la sección 4.3.2 es que el Meta-OS ha de contar con un sistema para la gestión del ciclo completo de vida (*Lifecycle*

management – LCM) de todos servicios desplegados por el orquestador del Meta-OS en varios niveles. El LCM implementado en este Meta-OS realmente aplica sobre los componentes de los servicios desplegados, puesto que son las entidades que equivalen a la unidad mínima de despliegue, y abarca desde el primer despliegue de un componente por parte de un usuario, pasando por su posible actualización (parámetros de configuración, reorquestación o cambio de nodo que lo ejecuta) hasta finalizar en el momento en el que es eliminado por un usuario. No obstante, en el LCM de este Meta-OS el ciclo vital de estos componentes no finaliza en este momento, ya que, cuando se elimina un servicio, aunque sus componentes (en forma de contenedores) se eliminan del nodo en el que se están ejecutando, la información de contexto relacionada con el mismo (es decir, las entidades NGSI-LD que almacenan los *context brokers*) no se elimina, manteniendo así cierta trazabilidad sobre dicho *ServiceComponent*.

Esta decisión está motivada por dos motivos: (i) habilitar el redespliegue de un servicio en un futuro aprovechando el acceso ubicuo a la información de contexto (ver 5.4.3.1); y (ii) dotar de un mayor nivel de observabilidad al Meta-OS, puesto que se mantiene un registro histórico de los servicios desplegados en el continuo. De manera adicional, también se ofrece la opción de eliminar permanente un servicio, es decir, borrar su registro de manera definitiva mediante la eliminación las entidades NGSI-LD de los CBs relacionados con el mismo.

Además, esta funcionalidad de LCM también permite que el estado de los componentes sea actualizado una vez ha finalizado el proceso de orquestación, dado que los diferentes LLOs monitorizan constantemente los contenedores desplegados para detectar fallos en su ejecución.

El LCM se refleja en los tres niveles que se muestran en la Figura 5.23:

- **Nivel global de Meta-OS** (Figura 5.24): el estado de los componentes se almacena en el atributo *serviceComponentStatus* de la entidad *ServiceComponent* del modelo de datos. Este estado es intrínseco al Meta-OS y puede ser consultado de manera global gracias a la ubicuidad de la información de contexto.
- **Nivel de LLO** (Figura 5.25)**:** reflejado en el atributo *status.conditions* de los *Custom Resources* de K8s manejados por los diferentes LLOs (ver 5.4.4). Incluye datos relacionados con ciertos procesos claves realizados por el LLO. Se puede consultar mientras el servicio no haya sido eliminado.
- **Nivel de contenedor** (Figura 5.26)**:** totalmente independiente al Meta-OS. Puede consultarse localmente mediante el sistema de gestión de contenedores del nodo. Por ejemplo, en el caso de K8s ejecutando el comando *kubectl get pod*.

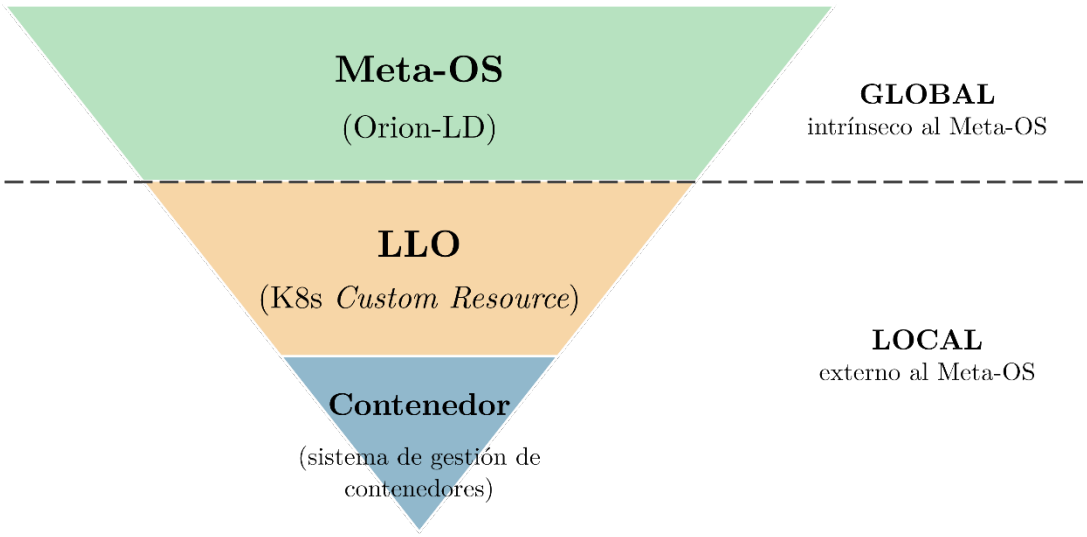


Figura 5.23: Niveles del LCM de los servicios desplegados en el Meta-OS

Service components:

ID	IE	Status	Container image	CLI args	ENV variables
influxdb	CloudFerro:fa163e5e25ef	Running	p4lik4ri/influxdb	×	
iperf-server	CloudFerro:fa163e5e25ef	Running	p4lik4ri/iperf-app	×	
iperf-client	CloudFerro:fa163e5e25ef	Running	p4lik4ri/iperf-app	×	
exp-orch	CloudFerro:fa163e5e25ef	Running	p4lik4ri/network-monitor	×	

Figura 5.24: Ejemplo del nivel global del LCM del Meta-OS reflejado en el portal de gestión

```
status:
  conditions:
  - lastTransitionTime: "2025-02-10T14:33:33Z"
    message: Joined to the domains network overlay successfully
    reason: ConfigurationComplete
    status: "True"
    type: NetworkingConfigured
  - lastTransitionTime: "2025-02-10T14:33:33Z"
    message: Service Component status updated in the Context Broker
    reason: StatusUpdated
    status: "True"
    type: ContextBrokerStatus
  - lastTransitionTime: "2025-02-10T14:33:33Z"
    message: Resource has been initialized successfully
    reason: CreationComplete
    status: "True"
    type: Initialized
  observedGeneration: 1
  status: Running
```

Figura 5.25: Ejemplo del nivel de LLO del LCM del Meta-OS reflejado en el CR de un componente

```
aeros-service-813-component-exp-orch-6dcbbdd7b5-gn8mx          2/2     Running     0          2m36s
aeros-service-813-component-influxdb-687b64b5cd-5nfdg          2/2     Running     0          2m37s
aeros-service-813-component-iperf-client-6f655bfd48-7jmr6      2/2     Running     0          2m36s
aeros-service-813-component-iperf-server-77b4484bdb-8djr5      2/2     Running     0          2m38s
```

Figura 5.26: Ejemplo del nivel de contenedor del LCM del Meta-OS reflejado en K8s. Primero se muestra el estado del *Pod* y después el estado de sus contenedores.

5.4.2.2. Conexión de servicios y componentes

En el sistema de orquestación y despliegue de servicios del Meta-OS los componentes que conforman los servicios son por defecto desplegados de manera distribuida en todos los nodos pertenecientes al continuo (a no ser que el usuario especifique lo contrario, como se detallará en el apartado 5.4.3.1), ya que el Meta-OS permite abstraer la complejidad intrínseca del continuo de computación y en consecuencia gestionarlo como si se tratara de una entidad uniforme. Este hecho da lugar a un escenario complejo en el que estos componentes se encontrarían totalmente distribuidos en el continuo, es decir, desplegados en distintos nodos de computación pertenecientes a diferentes dominios. Estos dominios, a su vez, están conectados a diferentes redes, ilustrando una vez más la heterogeneidad del continuo de computación. Por lo tanto, es necesario el desarrollo de una funcionalidad que permita abstraer esta heterogeneidad de redes para habilitar la conexión de los diferentes componentes desplegados en el continuo.

El doctorando está en condiciones de afirmar que el desarrollo de esta funcionalidad se trata de uno de los mayores desafíos técnicos que han aparecido durante la realización de esta tesis doctoral, puesto que no existe una solución clara y perfecta debido a su complejidad inherente. No obstante, se ha conseguido avanzar hasta conseguir la implementación de una solución completamente funcional, a partir de la cual avanzar para su futura mejora, ya que se ha identificado como una clara línea futura de investigación, y, por ende, listada en el punto 7.2. Se han definido una serie de requisitos para el desarrollo de esta solución:

- Debe ser independiente del componente o aplicación desplegada, permitiendo su uso en cualquier servicio de usuario gestionado por el sistema de orquestación del Meta-OS. Además, no debe interferir en su funcionamiento, operando de manera transparente.
- Ha de poder ejecutarse en forma de contenedor e interactuar con el contenedor principal del componente en cualquiera de los sistemas de gestión de contenedores.
- Debe ser lo menos invasiva posible para minimizar su impacto en el rendimiento del componente principal, especialmente considerando las limitaciones de recursos computacionales en algunos nodos *edge*.

- Ha de habilitar comunicaciones entre componentes desplegados en diferentes nodos y dominios para avanzar hacia la federación completa entre los dominios.

Teniendo en consideración esta serie de requisitos, se decidió tomar como base para la implementación la tecnología WireGuard (WG), ya que permite el establecimiento de túneles encriptados entre diferentes puntos de la red, con un bajo consumo de recursos y con una gran flexibilidad de configuración, como se detalló en el punto 2.5.1. Al mismo tiempo, se decidió descartar el uso de *libp2p,* tecnología utilizada para el establecimiento de las comunicaciones entre dominios (revisar 5.3.2), debido a que WireGuard es una tecnología más madura, al igual que el establecimiento de túneles en la capa de red en comparación con la creación de redes P2P a nivel de la capa de aplicación. Además, en el caso de *libp2p* sería necesaria la creación de un *stream* para cada protocolo de transporte (TCP, UDP ...) y la implementación de un sistema para redirigir el tráfico emitido por cada componente a través del *stream* correspondiente.

En este contexto, el componente EdgeMesh de KubeEdge ha logrado implementar un sistema basado en *libp2p* para el establecimiento de comunicaciones entre *Pods* desplegados en cualquier nodo de su clúster (ver 2.6.5.2). La arquitectura de este componente consiste en el despliegue de una instancia de su agente en cada nodo del clúster que se encarga de gestionar las conexiones entre *Pods*. Sin embargo, EdgeMesh se circunscribe únicamente a clústeres de K8s (es decir, un único dominio de aplicación con un solo sistema de gestión de contenedores), y cuyos nodos *edge* dependen de un nodo *cloud* centralizado que actúa como nodo *relay*.

La solución implementada para la conexión de los componentes y servicios consiste en el despliegue en cada uno de los dominios de una instancia de WireGuard (concretamente utilizando la imagen de contenedor *linuxserver/wireguard* [285]) junto con un servidor DNS ligero (Dnsmasq [286]) en forma de contenedores. Esta instancia de WireGuard cuenta con una interfaz de red *wg0* configurada por defecto (con sus claves pública y privada) que permite el enrutamiento de tráfico entre las interfaces de red internas del nodo en el que se despliega. La clave pública de esta interfaz de red se almacena en el *context broker* (atributo *publicKey* de la entidad *Domain*), para que pueda ser utilizada posteriormente por cualquier HLO del continuo para crear las conexiones entre componentes.

Cuando un usuario solicita la conexión entre componentes en el momento de la orquestación de un servicio, el elemento Deployment Engine del HLO que esté gestionando la petición de orquestación selecciona el servidor de WireGuard (de un dominio en concreto) que habilitará la conexión entre componentes. Seguidamente, se actualiza la configuración de esta instancia de WG (incluida en un archivo de configuración *.conf*) para crear en su interfaz *wg0* una subred única (si no existe, por ejemplo 10.13.5.1/24), que será utilizada para asignar la dirección IP interna del *peer* de WG correspondiente a cada componente, preparando su futura conexión al servidor de WireGuard. Además, también se actualiza el registro del servidor

DNS para asociar el nombre de los componentes con sus IPs internas dentro de la subred de WG. Esto permite que los componentes se comuniquen entre sí utilizando estos nombres en lugar de las direcciones IP.

```
[Interface]
Address = 10.13.10.2
PrivateKey = 2L8nX92FbIN8snTEb+bKleEIrPtyLXIiXhyE2ENxv0s=
DNS = 10.13.10.1
MTU = 1420

[Peer]
PublicKey = yL2vQPNLAe/ejZwtEW9z/L9PxlDdhX/QTXdckrF3sXM=
Endpoint = phd-domain.meta-os.eu:51820
AllowedIPs = 10.13.10.0/24
PersistentKeepalive = 25
```

```
[Interface]
Address = 10.13.0.1/24, 10.13.1.1/24, 10.13.10.1/24
ListenPort = 51820
PrivateKey = UE+7ifjWuYO4J43WlNGmdJaMhppaWJFB1skQzHm03ms=
PostUp = iptables -A FORWARD -i wg0 -j ACCEPT;
         iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT;
           iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

###START_BLOCK_Service:phd
[Peer] #component-1
PublicKey = knC3JxKFdiMwfUsAMyGupk8mZqf2rPLAwUhsoJ4kkBY=
AllowedIPs = 10.13.10.2/32

[Peer] #component-2
PublicKey = uuKFb/4irgjAnEcyPejWMBCJNAl/d2OceucdyaTPEUQ=
AllowedIPs = 10.13.10.3/32
###STOP_BLOCK_Service:phd
```

Figura 5.27: Configuraciones del cliente (componente) y del servidor (dominio) de WireGuard

En este punto, se ha establecido la configuración necesaria para habilitar la conexión entre componentes. A partir de esta información, cada componente ha de conectarse de manera individual a la instancia de WireGuard para hacerla efectiva. Al tratarse de contenedores, se ha optado por el despliegue de un contenedor adicional y específico para que actúe como cliente o *peer* de WG en el mismo *network namespace* del contenedor principal (correspondiente al componente o aplicación), de manera que la conexión sea transparente para este. El Deployment Engine incluye la información de conexión en el *Custom Resource* de K8s para que el LLO correspondiente la implemente. Los detalles de esta implementación se expondrán en el apartado 5.4.4 para cada tipo de LLO.

El mayor inconveniente de esta solución radica en el despliegue de un contenedor adicional por cada *ServiceComponent*, puesto que añade una carga extra de procesamiento que dependerá el uso de la red que haga el componente, siendo inicialmente muy reducida (sobre unos 10 MB de memoria RAM).

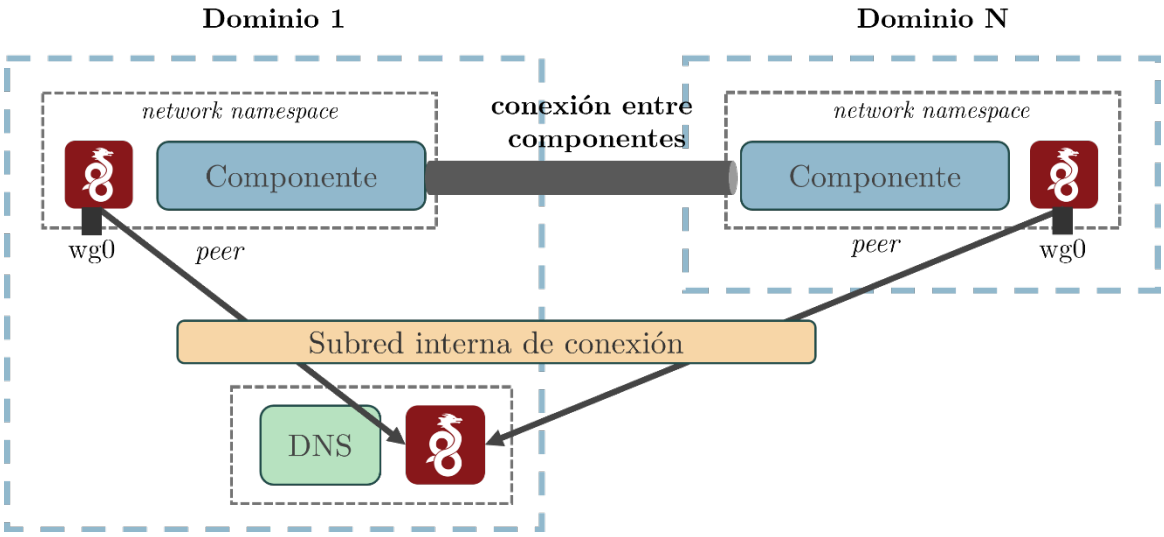


Figura 5.28: Conexión entre componentes desplegados en diferentes dominios

5.4.3. High-Level Orchestrator

El módulo de orquestación de alto nivel (HLO), descrito por primera vez en el apartado 4.5.2, se trata del bloque de entrada al sistema de orquestación del Meta-OS, debido a que recibe la petición de orquestación de un servicio por parte de un usuario y desencadena un proceso de orquestación individual para cada componente del servicio, que finaliza en la elección de un nodo para su despliegue y el envío de la orden de despliegue al LLO encargado de gestionarlo.

El desarrollo de este módulo toma como punto de partida el HLO desarrollado por el proyecto aerOS, dado que también sigue la arquitectura de orquestación definida por EUCEI. Esto es debido a que la dicho HLO fue diseñado (estructura, definición de los elementos internos, interacción entre los mismos, etc.) y desarrollado (utilizando el lenguaje Python) de manera colaborativa durante el proyecto, proceso en el que el doctorando formó parte activa y clave. No obstante, haber tomado como punto de partida esta implementación no significa que para esta tesis se haya utilizado esta herramienta en su actual estado y se haya desplegado directamente en el Meta-OS. En su lugar, el doctorando ha realizado modificaciones, adaptaciones y mejoras para que encaje con las características y requisitos definidos para el Meta-OS propuesto en esta tesis doctoral —pero sin dejar de lado el objetivo de que sea implementable en otros Meta-OS que siguen la arquitectura EUCEI— ya que a juicio del doctorando la implementación realizada en esta tesis puede aportar una serie de mejoras sobre el desarrollo original.

Esta nueva implementación del HLO está basada en el modelo de datos definido en la sección 5.2.1 y en la ubicuidad del acceso a la información de contexto (modelada utilizando este modelo de datos) detallada en la sección 5.2.3, sin dejar de lado la federación entre dominios, puesto que cada domino del continuo debe contar con su propia instancia del HLO. En resumen, las principales diferencias con el HLO que se ha tomado como punto de partida son las siguientes: (i) uso de NATS como *broker* de mensajería como reemplazo a Redpanda para mejorar la eficiencia en las comunicaciones asíncronas entre los elementos internos del HLO (ver 2.4.2); (ii) introducción del sistema de comunicaciones entre dominios que habilita su federación (detallado en el punto 5.3.2), puesto que en el desarrollo del proyecto de aerOS los HLO interactúan mediante peticiones directas a sus APIs, ya que están expuestas a través de API *Gateways* en IPs públicas, de manera que este nuevo enfoque sería más compatible con el paradigma de heterogeneidad que está presente de manera intrínseca en el continuo; (iii) introducción de tipos de despliegue adicionales, como el despliegue de una misma instancia de un servicio o componente en cada nodo perteneciente a una flota previamente definida, y la mejora del modo semiautomático (se describirá con más detalles en el siguiente subapartado 5.4.3.1); y (iv) preparación para el despliegue futuro de otros tipos de

virtualizaciones, como WebAssembly (revisar 2.7.2). Los detalles técnicos de esta última característica se detallarán en la sección 7.1.

No obstante, a pesar de las diferencias detalladas para contrastar la implementación realizada por el doctorando con la propuesta primigenia de aerOS, es relevante destacar nuevamente que el doctorando fue partícipe y protagonista de dicho diseño en el proyecto de investigación aerOS, situándole por tanto en una situación ideal para realizar cambios y mejoras que se adaptaran a los objetivos de esta tesis doctoral.

A continuación, se va a proceder a la descripción detallada de cada uno de los componentes internos que conforman el HLO. Tanto estos componentes como sus funcionalidades, las principales acciones que llevan a cabo y la interacción entre ellos se presentan tanto en la Figura 5.21 como en la Figura 5.29. Las comunicaciones entre estos elementos se realizan de manera asíncrona, utilizando un *broker* de mensajería NATS como habilitador común del mecanismo de publicación y suscripción de eventos. Por ejemplo, cuando se inicia el proceso de orquestación, el Elemento de entrada publica la información sobre el servicio en el *subject hlo.entry2aggregator* de NATS, que es recibida por el elemento Agregador de datos debido a que se encuentra suscrito al mismo *subject*. Como todos los elementos corresponden al mismo bloque funcional (HLO) y se espera que estén desplegados en el mismo nodo o en nodos adyacentes, se ha creado un único *stream* en NATS suscrito a todos los *subjects* relacionados con el HLO (utilizando el valor *wildcard hlo.**). En cada elemento interno se crea un *durable consumer* para suscribirse a su *subject* correspondiente, configurando una política de entrega de mensajes en la que se asegura exactamente un mensaje (*exactly-once*) a cada subscriptor. De esta manera, no se perderá ningún mensaje durante el proceso de orquestación de un servicio, aunque falle puntualmente alguno de estos elementos. En la sección 5.4.4.2 se ampliarán los detalles relacionados con las comunicaciones asíncronas realizadas a través del *broker* NATS, aprovechando su uso para los LLOs containerd y Docker.

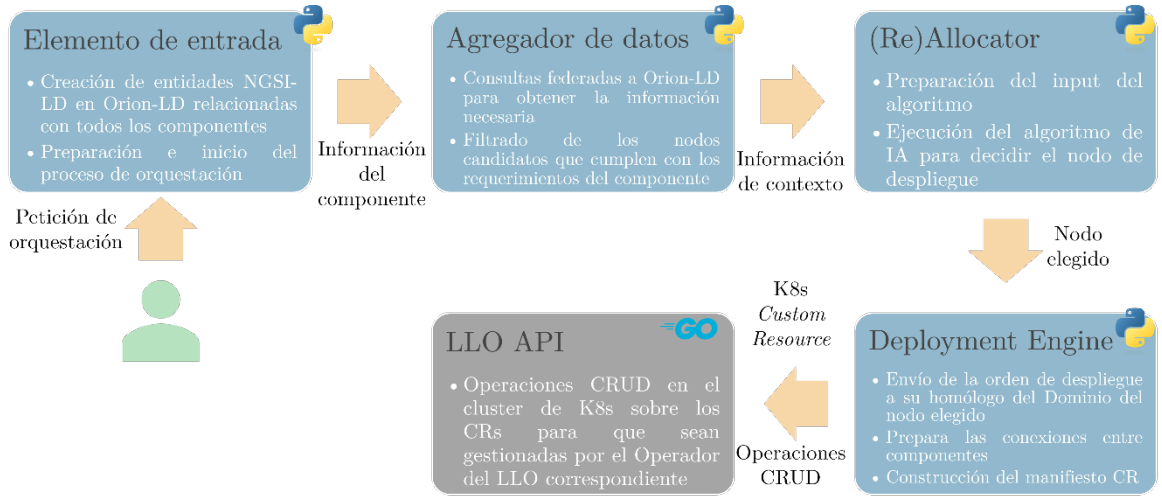


Figura 5.29: Diagrama de bloques del High-Level Orchestrator

En primer lugar, el Elemento de entrada recibe una petición de orquestación, ya sea desde la plataforma de gestión (vía el Entrypoint balancer) o directamente llamando a su propia API REST. Este elemento es el único que cuenta con una visión global del servicio, puesto que, a partir del transvase de información al Agregador, el proceso se realiza de manera individualizada para cada componente del servicio. La función principal de este componente es la introducción en el Meta-OS de la información relacionada con el servicio mediante la creación de las entidades NGSI-LD necesarias (entidades de los tipos *Service*, *ServiceComponent*, *NodeRequirements*, … de acorde con el modelo de datos definido) en el CB Orion-LD del mismo dominio —que son previamente construidas a partir de la información recibida. Seguidamente, publica un mensaje en el *broker* NATS para cada componente de dicho servicio con el identificador único de los mismos, de modo que sea recibida y procesada de manera individual y secuencial por el siguiente elemento del HLO: el Agregador de datos.

El Agregador de datos en primer lugar obtiene de Orion-LD todas las entidades relacionadas con el componente que está siendo procesado. Seguidamente, procesa dichas entidades para identificar los requerimientos de funcionamiento de dicho componente (RAM y CPU demandada, tipo de disco...). A partir de los atributos de la entidad *ComputingNodeRequirement*, realiza una consulta federada a Orion-LD (utilizando principalmente el parámetro *q* para realizar un filtrado) para obtener los potenciales nodos candidatos (entidades *ComputingNode*) para desplegar el componente. Esto es posible gracias al acceso ubicuo a la información de contexto del continuo con independencia del dominio desde el que se acceda, funcionalidad que ha sido reiterada en varias ocasiones debido a su carácter fundamental y transversal a todo el Meta-OS. Una vez obtenidos estos nodos, envía toda esta información (requerimientos de funcionamiento junto con los nodos candidatos) al elemento (Re)Allocator.

El módulo de toma de decisiones inteligentes, o (Re)Allocator, recibe la información previamente descrita y la adapta de acuerdo con un formato común contemplado por EUCEI, permitiendo que estos datos sean utilizados como datos de entrada del modelo de IA que tomará la decisión de despliegue de una manera inteligente. El objetivo de este formato común, tanto para la entrada como para la salida del modelo de toma de decisiones, es garantizar su flexibilidad y reemplazo sencillo (*plug-and-play*), de modo que los usuarios puedan integrar los algoritmos que consideren más adecuados según sus necesidades. Como se explicó en el apartado 4.5.2, el desarrollo de estos modelos queda fuera del ámbito de esta tesis doctoral, centrándose exclusivamente en su integración y uso dentro de la solución propuesta.

Finalmente, el Deployment Engine recibe la decisión tomada por el (Re)Allocator, es decir, el nodo en el que se desplegará el componente. Este elemento recopila la información detallada del nodo, junto con la información clave de despliegue, incluyendo la entidad LLO, que especifica el sistema de gestión de contenedores asociado con el nodo, así como las entidades relacionadas con el

componente. En caso de que el nodo seleccionado pertenezca a un dominio diferente, el Deployment Engine transfiere toda esta información a su homólogo de este dominio, permitiendo que el despliegue se realice de manera local en el dominio del nodo elegido. De este modo, el Deployment Engine prepara las conexiones del componente (especificadas en la definición de este) con otros componentes previamente desplegados o pertenecientes al mismo servicio (revisar el apartado anterior 5.4.2.2), con independencia del dominio y nodo en el que hayan sido desplegados. Además, se encarga de generar el *Custom Resource* de K8s en YAML, que depende del tipo de LLO asociado al nodo de despliegue, y cuyo formato se detallará en el próximo apartado 5.4.4. Finalmente, este módulo del HLO envía la orden de despliegue a la API general del LLO del dominio, para que realice la operación CRUD (*Create, Read, Update and Delete*) demandada en el clúster de K8s en los que han sido instalados los Operadores de K8s de los diferentes tipos de LLO soportados por el dominio.

5.4.3.1. Gestión de los diferentes tipos de despliegues

Hasta este momento se ha explicado el proceso estándar de orquestación de un servicio para que el lector se familiarice con él, puesto que se trata del proceso base a partir del cual se pueden realizar extensiones y adaptaciones. En relación con este aspecto, el sistema de orquestación ha sido diseñado para dotar de una mayor flexibilidad al usuario, habilitando configuraciones de despliegue adicionales que permitan flexibilizar el proceso evitando un posible encorsetamiento del proceso. Se ha decidido detallar estas adaptaciones dentro del apartado del HLO, puesto que es el elemento que implementa esta funcionalidad para finalmente enviar las órdenes de despliegue acordes al LLO, como si se tratara del proceso estándar de orquestación, sin diferencia alguna.

En primer lugar, el modo semiautomático de orquestación permite al usuario preseleccionar una serie de dominios y/o nodos de computación en concreto para que solamente sean tenidos en cuenta por el HLO, descartando los restantes. Además, el Meta-OS habilita la creación de subconjuntos de nodos bajo el criterio subjetivo del usuario, para identificar nodos pertenecientes al mismo fabricante, por ejemplo. Un nodo puede pertenecer a varios subconjuntos, y su pertenencia a los distintos subconjuntos se especifica en su atributo *deploymentFlavours* de su entidad NGSI-LD relacionada (ver el modelo de datos en la sección 5.2.1. Estos subconjuntos pueden ser utilizados para dos propósitos: i) la preselección de los nodos pertenecientes a un subconjunto para ser tenidos en cuenta en un proceso de orquestación semiautomático; y (ii) el despliegue de una misma instancia de un componente o de todos los componentes de un mismo servicio en todos los nodos pertenecientes al subconjunto, habilitando el despliegue por flotas en el Meta-OS.

En cuanto a los aspectos más generales, se añade la posibilidad de, si el nodo seleccionado pertenece a un clúster de K8s, considerar su clúster como el nodo

seleccionado y dejar en manos de K8s la elección del nodo. Asimismo, una de las configuraciones de orquestación más interesantes, consiste en seleccionar un componente de un servicio como principal (o simplemente considerar que se trata del que se introduce en primer lugar), y desplegar los demás componentes en el mismo nodo seleccionado para el componente principal, siempre que el nodo tenga suficientes recursos de computación disponibles. Esta opción es especialmente recomendable para servicios que consisten en una aplicación principal junto con una base de datos, ya que en la mayoría de las ocasiones no resultaría conveniente distribuir ambos componentes por el continuo.

Finalmente, gracias al LCM introducido en la sección anterior, los servicios que han sido eliminados (no permanentemente), es decir, el estado de sus componentes es *Finished*, pueden ser relanzados rápidamente (sin que el usuario haya de reintroducir sus parámetros de orquestación) debido a que su información es persistida en los *context broker*s. Este relanzamiento puede ejecutarse de dos maneras: (i) redespliegue de los componentes en los mismos nodos en los que se encontraban desplegados (siempre que los nodos cuenten con suficientes recursos); y (ii) ejecución del algoritmo inteligente de asignación de nodos para que sean desplegados en los nodos óptimos bajo su criterio.

5.4.4. Low-Level Orchestrator

El bloque de orquestación de bajo nivel, también descrito en el apartado 4.5.2, tiene como objetivo principal llevar a cabo de una manera tangible las órdenes de despliegue emitidas por el HLO, es decir, la creación, actualización o eliminación de un componente en un determinado nodo del continuo. Este componente tiene un ámbito totalmente local, enfocado al mismo dominio. Por lo tanto, debe haber tantos tipos de LLOs desplegados en un dominio como diferentes sistemas de gestión de contenedores presentes en los nodos que lo componen, estableciéndose una relación de 1:1, es decir, una única instancia de un LLO por un tipo de sistema de gestión de contenedores.

La existencia de diferentes tipos de LLOs hace imprescindible encontrar una solución generalista que permita definir un formato de datos y un modelo de funcionamiento común para todos ellos. Esto es fundamental, ya que las particularidades de cada LLO en cuanto a su interacción con el sistema de gestión de contenedores subyacente deben ser transparentes para el HLO dentro del proceso de orquestación. En otras palabras, el HLO es agnóstico al LLO utilizado para desplegar un componente en un nodo. Actualmente la literatura no ofrece soluciones específicamente diseñadas para abordar el propósito del HLO, lo que resalta el carácter innovador de esta tesis doctoral y su contribución en este ámbito. Del mismo modo, esta solución ha de ser necesariamente flexible para poder extenderla mediante el desarrollo de nuevos LLOs para que el Meta-OS consiga abarcar cualquier sistema de gestión de contenedores que sea requerido en el futuro.

Después de realizar un estudio de las tecnologías para la orquestación y el despliegue de contenedores (ver 2.6), así como de sistemas más generales de control del despliegue de aplicaciones, el *Operator pattern* de Kubernetes [166] fue identificado como la opción más prometedora sobre la que construir el marco de gestión de los LLOs. Estos Operadores de K8s pretenden emular las acciones de control que realizaría un operador o administrador humano experto sobre ciertas aplicaciones o servicios que se despliegan en una infraestructura, mediante el establecimiento de un sistema automático para controlar el ciclo de vida de estos recursos.

Entrando en los aspectos técnicos, los Operadores de K8s permiten extender la API de K8s incorporando conocimiento operativo mediante la creación de los llamados *Custom Resources* (CR), que son definidos utilizando la especificación OpenAPI como si de una API REST se trataran. En estos CRs se especifica el estado deseado del recurso al que aplican, y el elemento controlador ha de encargarse de realizar las operaciones necesarias, que pueden involucrar tanto recursos internos de un clúster de K8s (*Pods, Deployments, ConfigMaps, Secrets,* otros *CRs*...) como externos (llamadas a APIs, gestión de VMs...), para alcanzar el estado deseado previamente definido. Estas operaciones se realizan en el llamado *reconciliation loop* (bucle de reconciliación) del controlador del Operador [287].

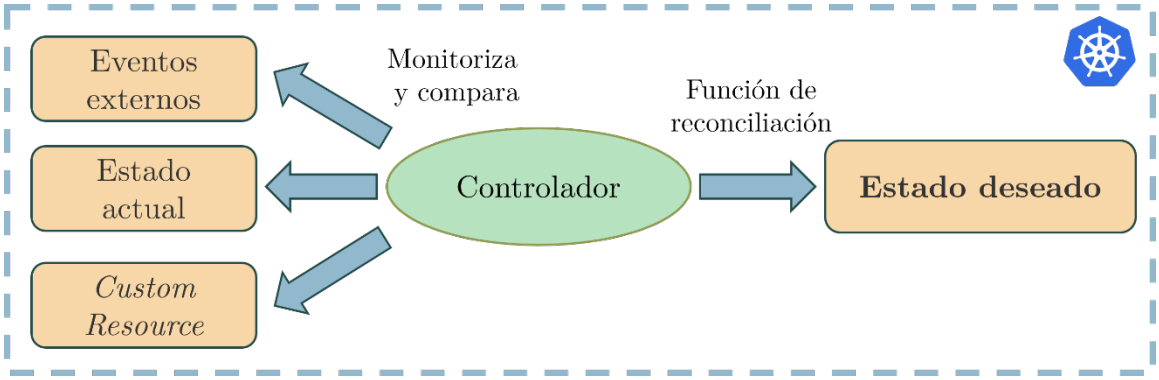


Figura 5.30: Diagrama de bloques de un Operador de K8s [287]

En el caso del bloque LLO de este Meta-OS, se ha definido una especificación común de un *Custom Resource* que aplica a todos los diferentes tipos de LLOs. La única diferencia entre ellos se establece en el parámetro *kind*, que indica el nombre del CR (por ejemplo, *ServiceComponentDocker*), y, por ende, el Operador específico que se encargará de reflejar el estado indicado en la definición del CR. Por consiguiente, se establece una relación única (1:1) entre un sistema de gestión de contenedores (por ejemplo, Docker) con un Operador (formado por un controlador y un CR) desarrollado específicamente para el mismo. Asimismo, se logra encapsular el funcionamiento interno del controlador, siendo transparente al funcionamiento completo del sistema de orquestación. Como se puede comprobar en la Figura 5.31, el nombre del componente sigue un patrón fijo: ***metaos-service-{serviceId}-component-{componentName}*** con el objetivo de identificarlos

inequívocamente. Esta nomenclatura se utiliza en todos los tipos de LLOs para nombrar el contenedor principal que despliegan, así como punto de partida para nombrar a los artefactos adicionales que necesiten crear.

```yaml
apiVersion: llo.meta-os.eu/v1alpha1
kind: ServiceComponentK8s
metadata:
  labels:
    app.kubernetes.io/name: component-nginx-of-the-service-123456789d
    app.kubernetes.io/instance: Service_123456789d_Component_nginx
    app.kubernetes.io/part-of: Service_123456789d
    app.kubernetes.io/managed-by: meta-os.eu
    app.kubernetes.io/created-by: LowLevelOrchestrator_Domain1_Kubernetes
  name: metaos-service-123456789d-component-nginx
spec:
  selectedNode:
    id: urn:ngsi-ld:ComputingNode:CloudFerro:4073c6a5a93b
    hostname: continuum-cluster-worker-1
  image: registry.gitlab.meta-os.eu/public/common-deployments/nginx
  exposePorts: true
  ports:
    - number: 80
      protocol: TCP
    - number: 443
      protocol: TCP
  persistence:
    mountPath: /usr/share/nginx/html
    hostPath: /mnt/data/metaos-service-123456789d-component-nginx
    accessMode: ReadWriteOnce
    size: 1024
  imageRegistry:
    # url: registry.gitlab.meta-os.eu
    username: user
    password: glpat-Fta1DNz6-aa_GQ6ywfpe
  networkOverlay:
    internalIp: 10.13.0.4/32
    dns: 10.13.0.1
    privateKey: WRTL8CmjwnyLkV9+Hk/u/iXGuB3C38i5BMTDlyD8wWA=
    publicKey: k8hAPelou4z1wBfyJO5cDUJutyuW10kOh5ezsh7p7SE=
    endpoint: domain.meta-os.eu:51820
    allowedIps: 10.13.0.0/24
```

Figura 5.31: Ejemplo de un *CustomResouce* para el LLO de tipo K8s

La principal desventaja con la que cuenta basar la solución de los LLOs en el Operator pattern de K8s es la necesidad de incluir un clúster de K8s en cada dominio, aunque su única función sea el despliegue de los Operadores y los CRs en el mismo para gestionar las órdenes de despliegue requeridas por el HLO. Asimismo, no se trataría de un requerimiento inaccesible, puesto que este clúster puede ser instalado utilizando alguna de las distribuciones ligeras de K8s descritas en la sección 2.6.5.2, cuyo consumo de recursos es limitado, al igual que el de los propios Operadores, haciéndola por tanto una aproximación factible para el continuo *Cloud-Edge-IoT*. Los nodos de este clúster no se computarían como nodos de computación del continuo.

En cuanto al funcionamiento del bloque LLO, el componente Deployment Engine del HLO realiza una llamada a la API del LLO para dar la orden de despliegue correspondiente (creación, actualización o eliminación de un componente). Esta API ha sido desarrollada utilizando Go y el cliente oficial de

K8s para este lenguaje con el objetivo de evitar que el HLO interactúe directamente con el clúster y abstraerlo de la complejidad de esta interacción. Además, permite aumentar la seguridad de las operaciones, puesto que se limita el acceso al plano de control del clúster de K8s del HLO, que podría ser externo al mismo. Seguidamente, la API crea, actualiza o elimina el *Custom Resource* en cuestión en el clúster, para que, dependiendo del *kind* del CR, se desencadene de manera automática la ejecución de la función de reconciliación del Operador correspondiente.

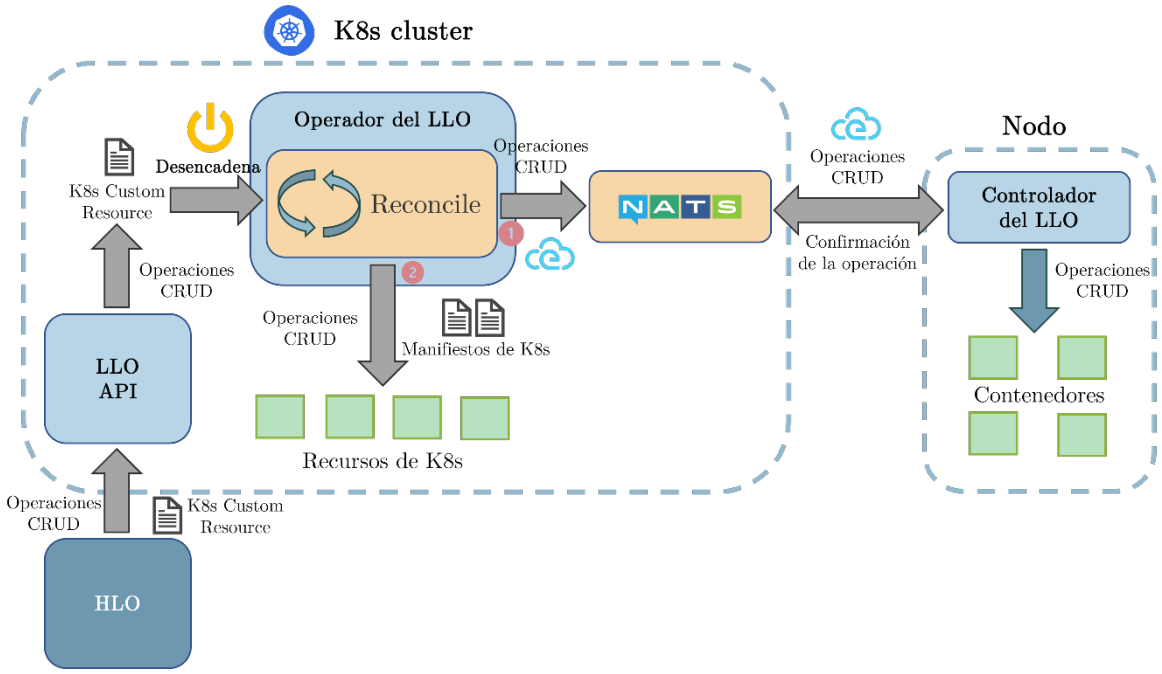


Figura 5.32: Diagrama de bloques general del Low-Level Orchestrator

Los Operadores del LLO se pueden clasificar en dos grupos: (i) el Operador del LLO de K8s, que realiza las operaciones directamente en el mismo clúster de K8s; y (ii) los demás Operadores que deben interactuar con un sistema de gestión de contenedores diferente (no K8s). Para este último grupo, se ha diseñado una arquitectura que consiste en el despliegue de un controlador específico en el nodo, de modo que interactúe con el sistema de gestión de contenedores del mismo para realizar las operaciones necesarias. Las comunicaciones entre el Operador (clúster de K8s) y el controlador del LLO (nodo de computación), se ha decidido que se realicen de manera asíncrona mediante el uso de un *broker* de mensajería NATS —que no se trata de la misma instancia utilizada por el HLO, siendo esta única para cada tipo de LLO—, utilizando un formato común de eventos que utiliza la especificación CloudEvents [288].

Para el desarrollo de todos los Operadores de K8s se ha utilizado el proyecto de la CNCF (en fase de incubación) Operator Framework, que proporciona un SDK para el desarrollo de Operadores de K8s utilizando el lenguaje Go (Operator SDK, que a su vez utiliza una herramienta oficial de K8s llamada Kubebuilder), junto

con un gestor del ciclo de vida de estos (OLM) [289]. Además, Operador Framework ofrece de manera abierta un repositorio de Operadores desarrollados utilizando este SDK de una manera totalmente integrada llamado OperatorHub [290].

En el marco de esta tesis doctoral, se han desarrollado únicamente los siguientes tipos concretos de LLOs, que corresponden con los sistemas de gestión de contenedores más usados: **K8s**, **Docker** y **containerd**, además de un LLO basado en **Balena,** utilizado como prueba de concepto para valorar la integración en el proceso de orquestación de servicios en nodos en los que se ha instalado un OS específico para la gestión de contenedores (revisar el apartado 2.6.4). Para este propósito se crearon dos proyectos base de Operadores, utilizando el CLI de Operator SDK, que corresponden con los dos grupos de Operadores de LLO definidos en los párrafos anteriores (K8s y no K8s). Estos proyectos no solo han sido ideados como base para la implementación actual, sino que también han sido diseñados para favorecer la extensibilidad del Meta-OS, permitiendo la incorporación de nuevos LLOs (por ejemplo, para los *container engines* Podman o iSulad) o la reestructuración de los existentes.

Además, todos los tipos de Operadores comparten la lógica inicial de la función de reconciliación en la que se detecta si el CR ha sido creado, actualizado o eliminado del clúster, así como una serie de procesos (escritura de mensajes de *log*, creación y asignación de las *labels,* gestión del *status* del CR, etc.), y configuraciones básicas relacionadas con la interacción de otros elementos del Meta-OS (conexión con Orion-LD, con el sistema de notificaciones, etcétera).

5.4.4.1. LLO para Kubernetes

El LLO de K8s podría considerarse como el LLO más simple en lo que respecta a su desarrollo, puesto que su Operador se limita a realizar operaciones en el mismo clúster en el que está desplegado, con las ventajas que ello comporta. Además, K8s ya proporciona una serie de funcionalidades específicas para la gestión de los contenedores, como su agrupación en *Pods*, la exposición de sus puertos mediante *Services* o la persistencia de información mediante *Persistent Volumes* (PV) y *Persistent Volume Claims* (PVC). Todo ello implica que la lógica del sistema se ejecuta íntegramente dentro del propio Operador, sin requerir elementos adicionales más allá del clúster de Kubernetes en el que se encuentra desplegado.

En primer lugar, después de realizar las configuraciones y conexiones oportunas, el Operador comprueba la existencia de una entidad NGSI-LD de tipo *LowLevelOrchestrator* correspondiente con él en el *context broker* local del dominio (por ejemplo, en este caso buscaría una entidad con un ID *urn:ngsi-ld:LowLevelOrchestrator:{DomainName}:Kubernetes*), y si no existe, la crearía.

En cuanto al proceso de despliegue de un componente de un servicio, en primer lugar, el Operador interpreta el contenido del CR, especialmente de su atributo *spec*. A partir de ésta, empieza a generar y crear los artefactos necesarios de K8s para finalmente desplegar el componente en forma de un *Deployment* con una única

réplica, que contiene un *Pod* (los artefactos mencionados anteriormente son usados por este *Pod*) formado por el contenedor principal correspondiente al componente, y, opcionalmente, un contenedor que actúe como cliente de la red de WireGuard para su conexión con otros componentes. Como en el caso de K8s se ha decidido que el elemento mínimo de computación sea un nodo del clúster (ver sección 4.2), este *Deployment* tiene asignado un *nodeSelector* para que el contenedor del componente se despliegue en el nodo en concreto elegido por el HLO (*kubernetes.io/hostname={selectedNode.hostname}).* No obstante, se deja abierta la opción a que este comportamiento sea modificado si así lo requiere el usuario (ver 5.4.3.1), eliminando este *nodeSelector* para que el plano de control de K8s decida en qué nodo es mejor desplegarlo bajo su propio criterio.

El Operador crea dos tipos de *Services* de K8s para exponer los puertos definidos: uno de tipo *ClusterIP* para los puertos que no se han de exponer al exterior, y otro de tipo *NodePort* para los que deben ser expuestos. Por otra parte, para la persistencia de información crea un *Persistent Volume Claim* (PVC) con la *StorageClass* configurada para ser usada por defecto en el clúster (marcada como *default*), que desencadenará la creación de un *Persisten Volume* por parte de K8s de manera automática. Además, si la imagen de contenedor del componente proviene de un repositorio privado, el Operador crea un *Secret* del tipo *kubernetes.io/dockerconfigjson* con las credenciales introducidas por el usuario, que es añadido en el parámetro *imagePullSecrets* del *Deployment* para que haga uso del mismo.

Respecto a la interconexión del componente, en el caso del LLO de K8s la conexión del contenedor del componente con la red de WireGuard se realiza mediante la creación de un *sidecar container* (llamados utilizando este término, son según la documentación oficial de Kubernetes, "contenedores secundarios que se utilizan para extender o mejorar la funcionalidad del contenedor primario de la aplicación sin afectar su código fuente, y que se ejecutan junto con el contenedor primario dentro del mismo *Pod*" [291]). Su uso está especialmente extendido en soluciones de *service mesh* (como Istio o Linkerd) en los que se añade un *proxy* ligero a los contenedores de los microservicios de una aplicación, de manera que se las comunicaciones con los *Pods* se establecen a través de este *sidecar container*, con el propósito último de establecer un plano adicional de observabilidad y de control sobre ellos.

Este *sidecar container* utiliza la imagen *linuxserver/wireguard* [285], una imagen creada por la comunidad LinuxServer.io que se trata de un cliente ligero de WG empaquetado como un contenedor, cuya configuración para conectarse a la red se realiza mediante un *ConfigMap* previamente creado por el Operador. Esta configuración está presente en el CR, y ha sido obtenida como resultado de los procesos que realiza el elemento *Deployment Engine* del HLO, como se explicó en el punto previo 5.4.3. De esta manera, el contenedor del componente puede hacer uso de la interfaz de red proporcionada por el contenedor cliente de WG para

establecer las conexiones necesarias con los otros componentes sin que sea realmente consciente de ello, como si de un *service mesh* se tratara.

Por otra parte, también es conveniente poner el foco brevemente sobre los procesos restantes: la actualización y la eliminación de un componente. La actualización de un componente se puede resumir en la modificación de ciertos parámetros del *Deployment* existente y de los demás artefactos creados, que incluso puede extenderse hasta la creación o eliminación de éstos si ya no son de utilidad. Por ejemplo, si un componente ya no necesita persistencia, se eliminan los PV y PVCs creados. En cuanto al proceso de eliminación, el Operador se limita a eliminar todos los artefactos relacionados con el componente creados en el clúster, de manera que solo quedaría registro del mismo en el *context broker* debido a la estrategia de LCM establecida (revisar 5.4.2.1).

Finalmente, para la monitorización del estado de los *ServiceComponents* (lo que equivale a los contenedores —recordemos que esta es una función que ha de realizar el LLO tal y como se especificó en el apartado 5.2.3.2), en el momento en el que se despliega un nuevo CR el Operador crea un *informer* (establece una conexión HTTP persistente del tipo *long polling* con el *kube-apiserver* de K8s para que informe sobre ciertos eventos) para el *Pod* y lo ejecuta en segundo plano mediante una *goroutine* para ser notificado cuando cambia el estado del *Pod* y actualizar en el *context broker* el estado del componente. En el caso que el mismo Operador fallara o se reiniciara, estos *informers* se perderían, puesto que no son persistidos. Por lo tanto, se ha incluido una función adicional para este propósito en las operaciones de inicialización del Operador.

5.4.4.2. LLO para Docker y containerd

La gran diferencia entre los LLOs pertenecientes a este grupo con el LLO de K8s reside en que las acciones finales para garantizar las órdenes de despliegue emitidas por el HLO no se ejecutan en el mismo nodo de computación o entorno que el Operador de K8s del LLO, sino que estas órdenes han de ser trasladadas a un elemento controlador que está desplegado en el nodo objetivo. Este hecho implica una serie de desafíos que se deben tener en consideración. El problema más importante es el establecimiento de un canal accesible, eficiente y confiable de comunicaciones entre ambos elementos (Operador y controlador), que asegure la entrega de los mensajes en casos en los que el nodo objetivo pierda la conexión o tenga algún fallo de funcionamiento, como puede ocurrir en entornos típicos del *edge computing*.

Para resolver este desafío se ha optado por utilizar comunicaciones asíncronas, concretamente mediante el uso de un *broker* de mensajería en cada dominio, puesto que este tipo de comunicaciones permiten que el Operador publique un mensaje con una cierta orden de despliegue, y que esta sea entregada al controlador del nodo al que está dirigida en el momento en el que sea posible, sin que se traduzca en una pérdida del mensaje. De la misma manera, se pueden establecer comunicaciones en

el sentido opuesto (desde el controlador hasta el Operador). Este sentido de las comunicaciones es realmente clave, puesto que en los tipos de LLO de este grupo, el Operador se limita a entregar un mensaje con una cierta acción al controlador, pero no tiene conocimiento del resultado de la misma, debido a que esta acción es realizada por el elemento controlador que se ejecuta en un nodo diferente, a diferencia del LLO K8s. Este canal permite al controlador publicar un mensaje con el resultado de la acción llevada a cabo por él (estado *Running* o *Failed*), habilitando la actualización del estado de funcionamiento del componente desplegado en el Meta-OS (LCM detallado en el apartado previo 5.4.2.1) por parte del operador.

Asimismo, el uso de un *broker* de mensajería permite establecer una relación jerárquica (1:N) entre el Operador y los controladores de los nodos de computación del dominio relacionados con este LLO, que a su vez es altamente escalable y es compatible con la pertenencia de nodos a redes privadas, ya que solamente el *broker* de mensajería, desplegado en el mismo nodo que el Operador, tiene que ser accesible por los mismos. Este tipo de arquitectura es similar a la de la mayoría de los sistemas de orquestación de contenedores adaptados al *edge computing* basados en K8s, como es el caso de KubeEdge, en el que solamente su parte *cloud* necesita ser accesible por los restantes nodos *edge* (ver 2.6.5.2). Aunque no solamente de contenedores, sino también de módulos Wasm, puesto que wasmCloud combina el uso de *brokers* NATS para establecer el canal de comunicaciones de su plano de control (ver 2.7.2.3).

El *broker* de mensajería elegido es NATS, al igual que en el HLO, debido a que es ligero y ofrece un sistema de persistencia de mensajes llamado JetStream, mediante el cual puede configurarse una política de entrega de mensajes en la que se asegura la entrega de exactamente un único mensaje a un subscriptor (llamada *exactly once*) [292]. Esta entrega se acaba realizando, aunque el subscriptor no esté disponible en el momento de la publicación del mensaje, debido a que se reintenta la entrega del mismo con una frecuencia periódica hasta que el suscriptor confirma su correcta recepción mediante el envío de un ACK (del inglés *acknowledgement*) al *broker*. Igualmente, JetStream habilita un registro de los eventos publicados en el *broker* para que quede constancia de los mismos y dotar de una capa de observabilidad adicional al sistema. También es necesario dotar de persistencia a JetStream para evitar la pérdida de la información en caso de que falle por algún motivo inesperado.

Concretamente, el Operador de este grupo de LLOs ha de crear un *stream* de JetStream, que actúa como un almacén de mensajes, en el *broker* NATS que únicamente será utilizado para este tipo de LLO en el dominio. Este *stream* contiene una serie de metadatos (ID del LLO, tipo de sistema de gestión de contenedores...) y está suscrito al *subject events.{LLO-ID}.**, puesto que el controlador de cada nodo publicará en una serie de *subjects* propios *events.{LLO-ID}.{Node-ID}*.

En cuanto a los controladores, para poder consumir los mensajes han de crear un *durable consumer* nombrado como el ID del nodo sobre el que actúan, filtrando

los mensajes por el *subject events.{Node-ID}* y con una política de ACKs explícitos, es decir, la recepción de cada mensaje necesita ser confirmada para que el *stream* lo marque como entregado y no insista en su reenvío.

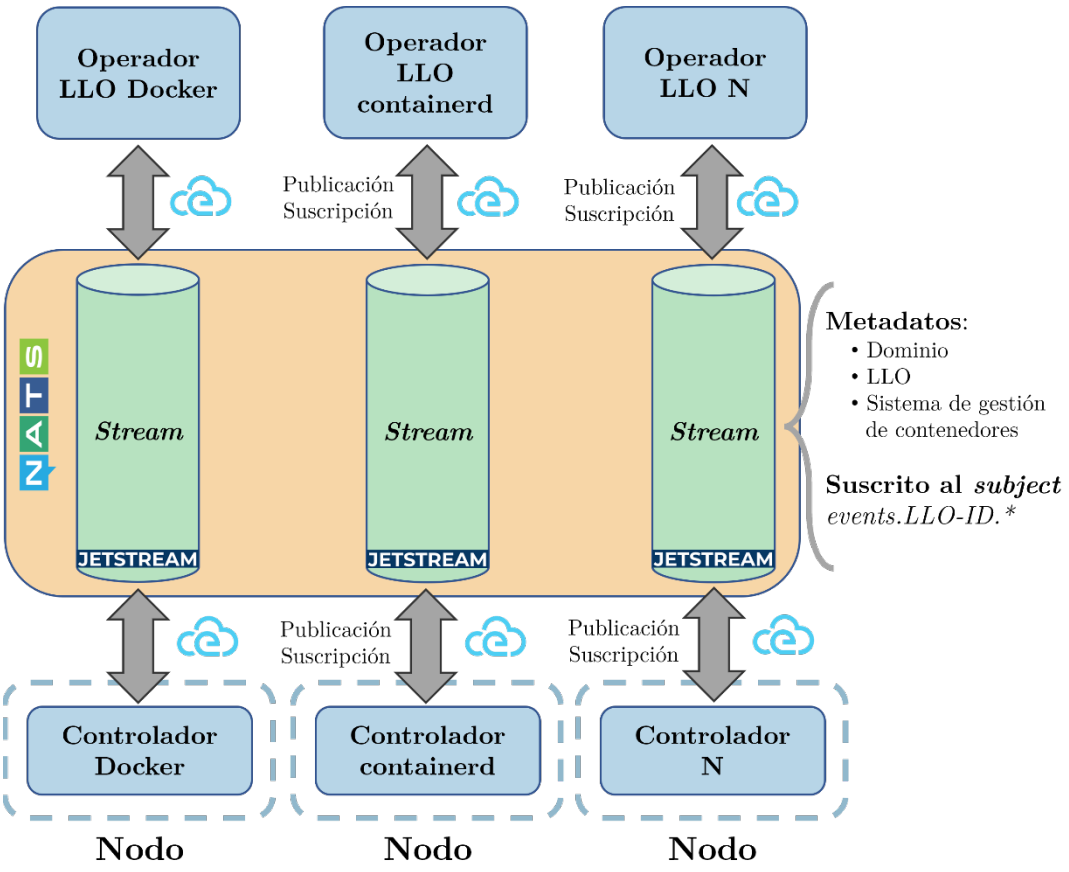


Figura 5.33: Estructura de las comunicaciones entre los Operadores y controladores de los LLOs a través del *broker* NATS

Cuando una arquitectura se basa en el uso de *broker* de mensajería, no solo basta con definir una estructura y nomenclatura de *topics* o *subjects* en los que se estructura la publicación de mensajes. También aparece la necesidad de definir un formato común de mensajes para evitar que cada elemento publicador use su propio formato, además de fomentar la extensión de elementos suscriptores, sirviendo en el ámbito de esta tesis doctoral para fomentar la extensión de los tipos de LLO disponibles. Los mensajes publicados en NATS han de seguir un formato muy concreto llamado CloudEvents, que se trata de una especificación para estandarizar la información sobre eventos bajo el paraguas de la CNCF, inicialmente pensado para ser usado en ecosistemas basados en *serverless* (ver punto 2.6.6). Además, CloudEvents proporciona una guía, junto con una librería propia, para mapear este formato de eventos a los mensajes propios de NATS [288]. Este mapeo consiste en el uso del formato JSON con una estructura concreta en la que se incluyen ciertos metadatos (versión de la especificación, formato del contenido, marca temporal del mensaje, etc.) y el contenido real del mensaje dentro del atributo *data*. En cuanto

al formato en concreto del contenido, también ha sido creado desde cero, pero tomando claramente como inspiración el formato común de los CR de los Operadores LLO (Figura 5.31), aunque utilizando JSON como formato. La mayor novedad es la inclusión del atributo *action* (*create*, *update* o *delete*) para definir la operación CRUD que ha de ejecutar el controlador.

```
------------------ Message ------------

Subject: mySubject

------------------ payload ------------

{
    "specversion" : "1.0",
    "type" : "com.example.someevent",

    ... further attributes omitted ...

    "data" : {
        ... application data ...
    }
}

---------------------------------------
```

Figura 5.34: Especificación de CloudEvents para NATS junto con un ejemplo de un mensaje enviado al controlador

Una vez realizada la explicación de las comunicaciones entre el Operador y los controladores de los nodos, es necesario poner el foco en el elemento controlador de este grupo de LLOs. Este elemento actúa como cliente de dos elementos clave para su funcionamiento: (i) el *broker* de mensajería NATS, para recibir órdenes de despliegue por parte del Operador; y (ii) el sistema de gestión de contenedores del nodo en el que se ejecutan, para poder realizar operaciones relacionadas con los contenedores, requiriendo los permisos necesarios a nivel de sistema operativo. Estos dos requisitos, junto con la necesidad de ser desplegado en una gran variedad de nodos de computación, hacen del lenguaje Go la opción perfecta para su desarrollo, puesto que cuenta con una librería cliente para NATS, librerías cliente para la mayoría de los sistemas de gestión de contenedores (Docker, containerd...) y es fácilmente compilable para diversas arquitecturas de CPU.

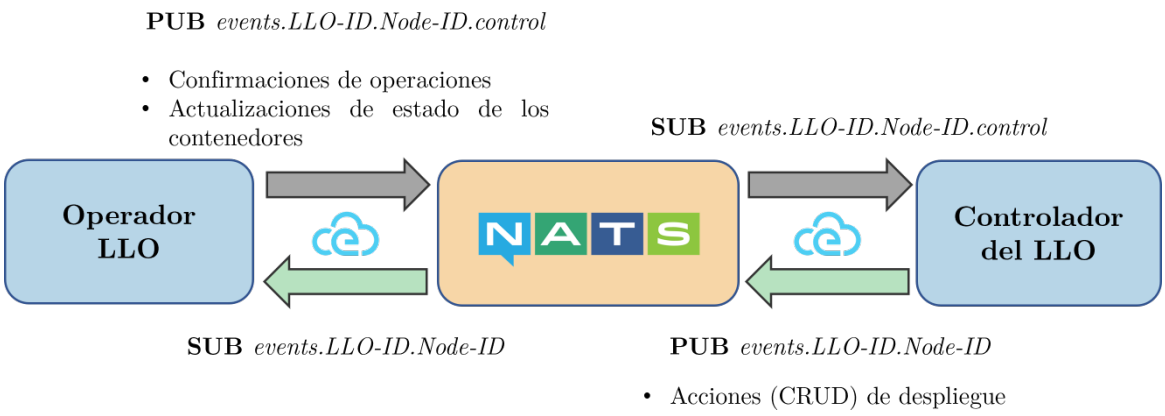


Figura 5.35: Comunicaciones detalladas entre el operador y el controlador de un LLO mediante NATS

LLO Docker

En este punto se va a detallar el funcionamiento concreto del LLO Docker, evitando repetir los procedimientos comunes con el Operador de K8s. Cuando este Operador recibe la orden de despliegue del componente de un servicio, interpreta el contenido del CR para preparar con toda su información el mensaje que ha de enviar al elemento controlador para que realice el despliegue. En este momento el Operador actualiza el estado del componente a *Deployed*, para informar de que el proceso de despliegue del componente ha sido gestionado correctamente por el LLO, pero aún no ha finalizado.

En el momento que el controlador recibe el evento mediante el *durable consumer* previamente creado, confirma su recepción enviando un ACK doble para asegurar la integridad de las comunicaciones, es decir, el *broker* NATS también ha de confirmar que ha recibido el ACK. Seguidamente procesa el mensaje ejecutando la función *handler* del *consumer*, en la que en primer lugar se identifica el proceso especificado en el atributo *action* del mensaje. En el caso de la creación de un componente, este se despliega en forma de un único contenedor, exponiendo los puertos requeridos, además de crear previamente los volúmenes necesarios. Si el componente necesita comunicarse con otros componentes, previamente a la creación y ejecución del contenedor, se crea en primer lugar un contenedor para que actúe como cliente de WireGuard (misma imagen que en el LLO de K8s, *linuxserver/wireguard)* al que se introduce la configuración para conectarse a la red con un volumen de Docker, que es necesario crear antes de que se ejecute. Por último, se despliega el contenedor del componente de manera que comparta el *network namespace* con el contenedor de WireGuard. Aquí radica una de las mayores diferencias con K8s, puesto que, si es necesario exponer puertos del contenedor del componente a nivel del nodo, se ha de hacer sobre el contenedor de WireGuard y no directamente sobre el contenedor del componente.

Por otra parte, también es conveniente poner el foco brevemente sobre los procesos restantes: la actualización y la eliminación de un componente. En el caso de la actualización, Docker no permite actualizar contenedores que ya han sido creados, por lo que la única alternativa consiste en la eliminación del contenedor actual para posteriormente crear un nuevo contenedor de acorde con los nuevos requerimientos del componente. En cuanto al proceso de eliminación, el controlador se limita a eliminar todos los recursos creados relacionados con el componente, de manera que solo quedaría registro del mismo en el *context broker* debido a la estrategia de LCM establecida (revisar 5.4.2.1).

Finalmente, para la monitorización del estado de los contenedores, el controlador hace uso del sistema de notificaciones de eventos en tiempo real que incorpora Docker llamado *docker system events*. Este elemento crea una política de filtrado para obtener solamente los eventos relacionados con la primera ejecución y los posibles fallos de los contenedores que gestiona como parte del Meta-OS. Cuando el Docker Engine notifica al controlador sobre alguno de estos eventos, el controlador lo publica en el *subject events.{LLO-ID}.{Node-ID}.control*, al que está suscrito el Operador del LLO de Docker. Por último, el Operador actualiza el estado del componente en el *context broker*.

LLO containerd

El funcionamiento general del LLO para containerd es prácticamente idéntico al de Docker, por lo que solo se van a describir los detalles relacionados con la interacción del elemento controlador con containerd para la creación de los elementos necesarios para la ejecución del contenedor. En este tipo de LLO esta implementación se puede considerar como más desafiante, puesto que containerd se trata de un *container runtime* de alto nivel (ver 2.6.2 y 2.6.3) que usualmente interactúa con un elemento de abstracción superior, como un *container engine* (Docker) o una plataforma de orquestación de contenedores (K8s). Por lo tanto, ciertas acciones, como la creación de volúmenes, requieren la realización de ciertas acciones adicionales que serían abstraídas en el caso de usar un *container engine* en lugar de un *container runtime* de manera directa. Por ejemplo, containerd no cuenta con una abstracción para la creación de volúmenes, por lo que la carpeta tiene que ser creada manualmente en primer lugar en el *host* para después ser montada en el contenedor y que pueda hacer uso de ella.

No obstante, el hecho de no instalar dicho *container engine* en un nodo se traduce en un ahorro en los recursos de computación, que puede ser utilizado para desplegar servicios o componentes adicionales, o mejorar su desempeño. Esta es sin lugar a duda el motivo principal para el desarrollo del LLO de containerd, encajando perfectamente en entornos típicos del *edge computing* del continuo y permitiendo la integración de un mayor tipo de nodos de computación en el Meta-OS. A su vez, debido a que containerd (a diferencia de K8s o Docker) está disponible de manera nativa para la arquitectura de CPU de código abierto RISC-V, se decidió compilar el LLO de containerd para esta arquitectura. Esta compilación fue posible

gracias a la flexibilidad varias veces comentada del lenguaje Go para la compilación de desarrollos en binarios ejecutables para múltiples arquitecturas. Aunque RISC-V aún no está suficientemente implementado en la industria, varios agentes tecnológicos, especialmente asiáticos y europeos (incluyendo a la misma EC), han empezado a fijarlo como un objetivo clave debido a su licencia abierta, viéndolo como un primer paso hacia el desarrollo de *hardware* sin el pago de licencias y la consiguiente unión *hardware-software open source*, un anhelo que siempre ha sido perseguido en el ecosistema IT. Este aspecto se extenderá en el capítulo 7.

5.4.4.3. LLO para balenaOS

El LLO Balena se trata de un tipo de LLO muy concreto, puesto que solo está diseñado para ser desplegado en nodos de computación que usen Balena como su sistema operativo, es decir, una serie limitada de dispositivos embebidos o SBCs como Raspberry Pi, Orange Pi, ASUS IoT Tinker o NVIDIA Jetson, así como ciertos modelos de mini PCs como INTEL NUC. No obstante, en este LLO solamente se contempla el uso del OS de Balena (balenaOS) en particular, de manera que no se tienen en cuenta sus funcionalidades relacionadas con el control de dispositivos y despliegues de aplicaciones en ellos de manera totalmente centralizada mediante el uso de su plataforma de gestión *cloud*, ya sea en su versión PaaS (Platform as a Service) de pago por uso o su versión gratuita y abierta OpenBalena. El uso de esta plataforma queda claramente fuera del ámbito de la solución aportada por esta tesis doctoral.

La gran ventaja de Balena consiste en que su *container engine* interno (balenaEngine [114]), que viene instalado de manera intrínseca en balenaOS, es un *fork* o adaptación del Docker Engine especialmente realizada para su OS y ecosistema en general. Por consiguiente, tanto el código fuente (la librería cliente de Docker para el lenguaje Go puede ser utilizada para este *container engine*) como la arquitectura del LLO Docker (basado en un modelo asíncrono de publicación y suscripción utilizando un *broker* NATS) pueden ser reutilizados completamente, sin necesidad de cambiar su funcionalidad interna. Este supuesto ha sido comprobado cuidadosamente, directamente sobre nodos con este OS, puesto que en su documentación oficial se afirma que puede no tener compatibilidad con Docker para ciertas funcionalidades. Es más, la versión del Docker Engine sobre la que está construida la versión del balenaEngine más reciente durante el desarrollo de este componente (20.10.43) es bastante antigua, ya que la última versión de Docker es la 28.0.0. La única diferencia entre ambos controladores se encuentra en la configuración de la localización del *socket* para conectarse con el *Balena engine* (*/var/run/balena-engine.sock*), que es necesaria para que el cliente de Docker pueda interactuar con este *container engine*.

Otra particularidad del controlador de este LLO es que la imagen base de contenedor ha sido construida sobre una arquitectura CPU ARM64 (o ARMv8), y no la típica AMD64 (x86_64) que utilizan la mayoría de PCs o servidores, puesto

que la mayoría de los dispositivos compatibles con el balenaOS son SBCs que utilizan esta arquitectura de CPU.

En caso de que las acciones realizadas por el LLO Docker no hubieran sido compatibles con el *Balena engine*, se contempló como alternativa el desarrollo de una solución que consistiría en el despliegue de un controlador en un nodo de computación cercano y con acceso directo al nodo que realmente tiene instalado balenaOS. De esta manera, este controlador se trataría del módulo que realmente que interactuaría con el *broker* NATS y traduciría los eventos recibidos en comandos del *Balena CLI* para ejecutar las órdenes de despliegue en el nodo final. Sin embargo, esta solución sería más limitada y conllevaría asociada la posible pérdida de la capacidad de monitorización del estado de los contenedores desplegados.

5.4.4.4. Comparación de despliegue de recursos

Para concluir con el bloque Low-Level Orchestrator del sistema de orquestación del Meta-OS, el doctorando ha decidido crear una tabla en la que se resumen las diferentes maneras en las que se plasman las acciones claves del proceso de despliegue del componente de un servicio en los diferentes tipos de LLOs desarrollados. De esta manera el lector también puede observar de una manera más ilustrativa los detalles más técnicos de este componente.

Tabla 5.1: Comparación de acciones clave en los diferentes tipos de LLOs

Acción	Kubernetes	Docker/Balena	containerd
Despliegue del componente	*Deployment* formado por una sola réplica del *Pod* que contiene el contenedor del *ServiceComponent*	Contenedor de Docker	Contenedor de containerd
Actualización del componente	Actualización del *Deployment*	Eliminación del contenedor actual y posterior despliegue de un nuevo contenedor	Eliminación del contenedor actual y posterior despliegue de un nuevo contenedor
Exposición puertos	Despliegue de dos servicios de K8s: un servicio interno del tipo *ClusterIP* y un servicio externo del tipo *NodePort* (identificado por el sufijo "*-ext*")	Exposición de los puertos solamente en la red interna del contenedor o exposición directa a nivel del nodo para los puertos externos.	Exposición de los puertos solamente en la red interna del contenedor o exposición directa a nivel del

			nodo para los puertos externos.
Persistencia	Creación de un PVC utilizando la *StorageClass* configurada por defecto en el clúster, que crea a su vez un PV asociado.	Creación de un volumen de Docker local en el nodo.	Creación manual de una carpeta en el nodo para que después sea montada en el contenedor.
Credenciales de un repositorio privado	Las credenciales se guardan en un *Secret*, que es utilizado en el *imagePullSecrets* del *Deployment*.	Las credenciales solamente se utilizan para descargar la imagen y no se persisten más allá.	Las credenciales solamente se utilizan para descargar la imagen y no se persisten más allá.
Networking	Creación de un *sidecar container* dentro del mismo *Pod* del componente. La configuración del cliente de WG se configura utilizando un *ConfigMap*.	Creación de un contenedor que comparte el mismo *network namespace* con el contenedor del componente. En el caso de que existan puertos externos, este contenedor ha de exponer los puertos sobre la máquina.	Creación de un contenedor que comparte el mismo *network namespace* con el contenedor del componente. En el caso de que existan puertos externos, este contenedor ha de exponer los puertos sobre la máquina.
Arquitecturas de CPU soportadas	x86_64, ARMv8	x86_64, ARMv8	x86_64, ARMv8, RISC-V

5.5.Plataforma de gestión unificada

La plataforma de gestión unificada es el único punto de entrada para la gestión visual (interfaz de usuario — UI) del continuo de computación a través del Meta-OS desarrollado en esta tesis doctoral. Al existir en un único punto, su implementación en un continuo está presente únicamente en el dominio seleccionado como *entrypoint*, pero a su vez es fácilmente migrable a otro dominio

del continuo en caso cuando la situación lo requiera. Esta plataforma se ha desarrollado como una aplicación web, concretamente como una SPA (*Single-Page Application*), compuesta por dos elementos: una parte *frontend* (la propia aplicación web con la que interactúan los usuarios) que intercambia constantemente información con la parte *backend*, que se encarga de obtener los datos o realizar las peticiones necesarias en cada momento a ciertos componentes elementales del Meta-OS, como el *context broker* o el HLO.

Esta división, que es bastante frecuente en el ámbito del desarrollo de aplicaciones web o portales de gestión con los que los usuarios deben interactuar constantemente, se ha realizado para descargar a la aplicación web (el *frontend*) de la ejecución de ciertas funcionalidades que demandan más recursos de computación (procesado de datos, gestión del ciclo completo del sistema de notificaciones, interacción directa con otros componentes, etc.), con el objetivo de reducir su tamaño, así como mejorar su eficiencia, dado que se establece una clara división de responsabilidades. Por lo tanto, siguiendo esta arquitectura, la parte *backend* actúa en este caso como un *middleware* entre la aplicación web y las APIs REST pertenecientes a los componentes elementales del Meta-OS. Además, permite evitar algunos problemas de seguridad frecuentes, así como lidiar con el CORS (*Cross-Origin Resource Sharing*, un mecanismo de seguridad que permite o restringe solicitudes HTTP entre diferentes orígenes en un navegador), que suele ser un quebradero de cabeza en este tipo de desarrollos.

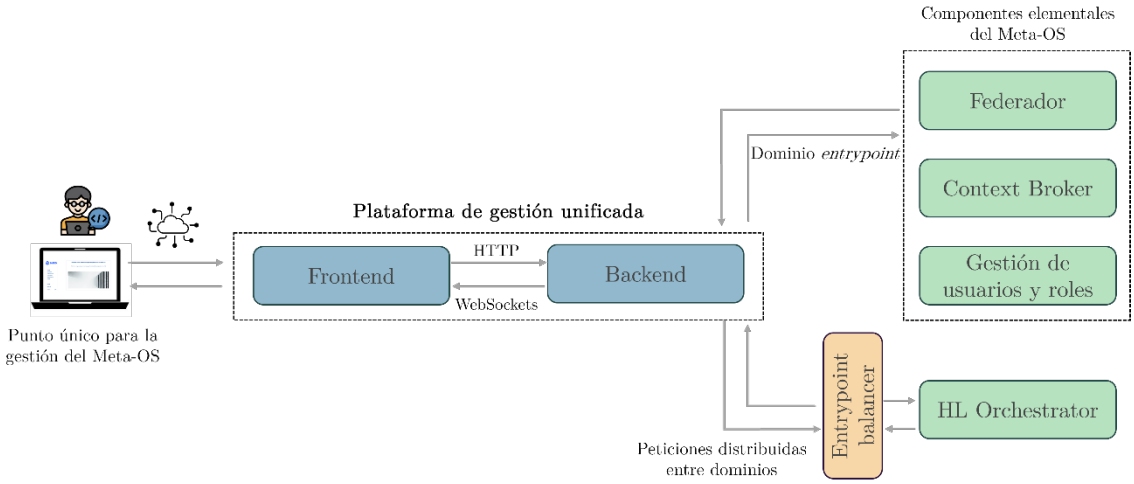


Figura 5.36: Diagrama de bloques de la plataforma de gestión unificada

Queda claro que el portal o plataforma de gestión debe ser utilizado por una serie de usuarios o personas físicas. Por este motivo, queda patente la necesidad de integrar algún sistema de seguridad o base de datos para la gestión de la identidad de los mismos, y de esta forma permitir solamente el acceso a los usuarios autorizados. Este sistema debe incluir funcionalidades relacionadas con la gestión de los propios usuarios y la asignación de permisos o roles a estos para definir las acciones que pueden realizar en el portal, y, por ende, en el continuo. En este caso se ha optado por utilizar una única instancia de la herramienta Keycloak, que se

trata de un gestor de identidades o IdM [293], con un espíritu totalmente funcional, de manera que no aporta ninguna novedad tecnológica al Meta-OS desarrollado. Esta decisión se tomó puesto que la ciberseguridad fue calificada como un aspecto transversal al sistema desarrollado en esta tesis, de manera que no se ha abordado específicamente, aunque esto no significa que no se haya tenido en cuenta en varios niveles, como se razonará en el apartado 5.7.

Keycloak utiliza el protocolo OAuth 2.0 para la autorización de acceso a los usuarios registrados en su base de datos. La aplicación web utiliza el flujo de OAuth 2.0 llamado *Authorization Code Flow with Proof Key for Code Exchange* (PKCE) [294], mediante el cual se redirige a un usuario no identificado a una pantalla en la que debe introducir sus credenciales, que son enviadas a la API de Keycloak para que compruebe la validez de estas, y en caso de ser válidas, redirige al usuario a la página de bienvenida de la aplicación web. Además, Keycloak genera un *token* JWT (JSON *Web Token*) que contiene información clave del usuario, entre la que destaca el rol con el que ha sido asignado, que es enviado a la aplicación web para que lo gestione de acuerdo con su funcionamiento interno. Este token se reutiliza para proteger las comunicaciones entre los dos componentes (*frontend* y *backend*) y evitar el uso del *backend* de manera directa por parte de usuarios o agentes no autorizados para ellos.

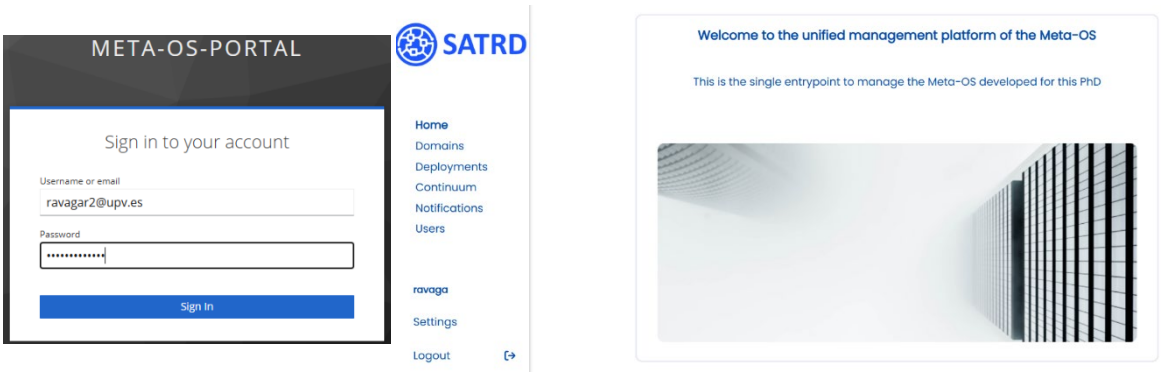


Figura 5.37: Páginas de *login* y de bienvenida de la plataforma de gestión

La implementación técnica de la aplicación web (la parte *frontend*) se realizó utilizando el lenguaje Vue.js principalmente junto con la herramienta para el almacenamiento y gestión del estado Pinia. Mientras que el doctorando fue responsable del diseño estructural del portal, definiendo la organización de las páginas, vistas y contenidos, el aspecto puramente visual y estético de la aplicación web fue desarrollado por un equipo especializado en diseño de interfaces y experiencia de usuario. A la hora de ser empaquetada para su despliegue en forma de contenedor, esta aplicación web es servida por un servidor web nginx, que a su vez actúa como un *reverse proxy* para habilitar las comunicaciones con el *backend*.

En cuanto a la parte *backend*, se trata principalmente de una API REST desarrollada en Java y utilizando Spring Framework 6, junto con Spring Boot, Spring Security y Spring Cloud [295]. Una de las características más importantes a

destacar es que este *backend* no interopera con una base de datos o un sistema de almacenamiento para guardar ciertas configuraciones o su estado, la cual es una práctica común en los paneles de administración desarrollados como aplicaciones web. Esta decisión de implementación se tomó para afianzar la movilidad de la plataforma de gestión en el continuo, habilitando su capacidad de migración entre dominios que se ha mencionado previamente.

Este *backend* integra un canal de comunicaciones basado en el protocolo WebSockets entre el *frontend* y el mismo para habilitar el intercambio de mensajes en tiempo real, como las notificaciones que se muestran a los usuarios en la plataforma de gestión. Este canal ha sido desarrollado utilizando Socket.IO, que se trata de una herramienta que permite el establecimiento de comunicaciones con baja latencia, bidireccional y basada en eventos, yendo un paso más allá de los WebSockets tradicionales [296]. Además, estas notificaciones se persisten en los *context brokers* del continuo para poder ser consultadas posteriormente, y utilizan el formato de datos *Alert* de FIWARE Smart Data Models [297].

Una vez se han descrito las características y la implementación técnica de la plataforma de gestión unificada, se va a realizar un pequeño recorrido por las diferentes páginas y vistas que lo componen. En primer lugar, en la página de los dominios del continuo, se muestra una tabla con información básica de los mismos, que permite acceder a sus detalles para presentar las características de cada uno de los nodos de computación que los componen.

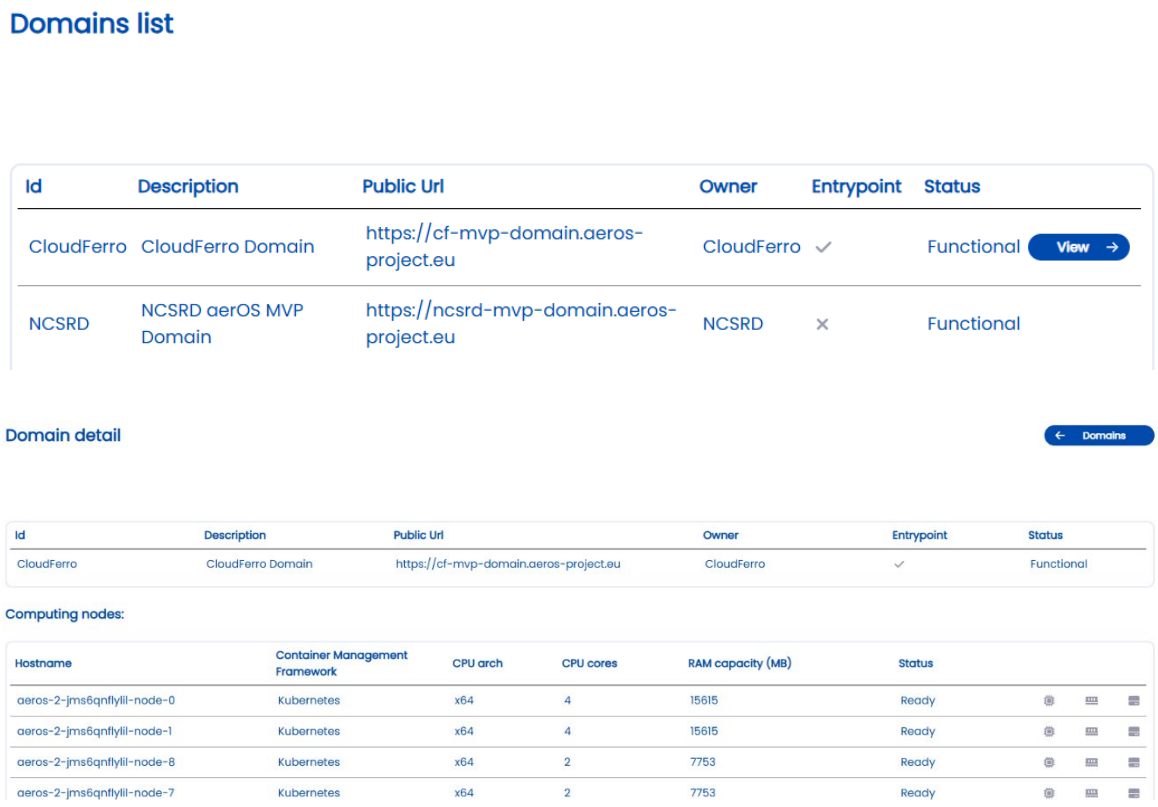
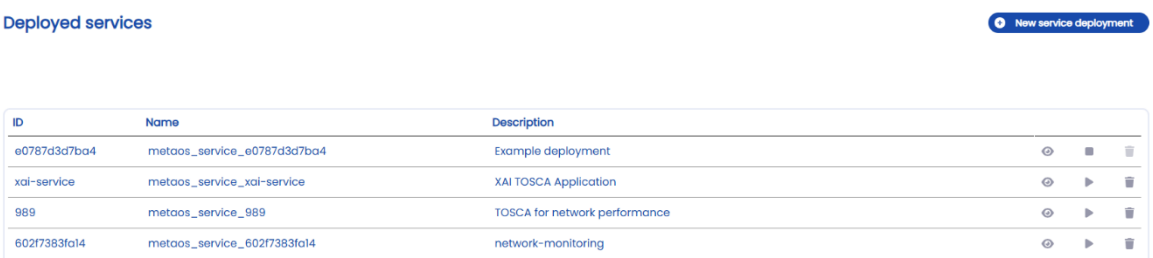


Figura 5.38: Vista de la lista de dominios y sus detalles

Esta misma información se presenta de manera más visual en forma de un gráfico de tipo árbol (realizado utilizando la librería *v-network-graph* [298]), el cual pretende representar la topología de cada uno de los dominios.

Figura 5.39: Vista de la topología de los dominios

La vista de los servicios desplegados tiene una estructura similar a de los dominios del continuo. Los servicios desplegados en el continuo se plasman en una tabla, junto con una serie de acciones que pueden realizar los usuarios. Si el servicio ha sido desplegado y sus componentes están siendo ejecutados en los nodos de computación asignados, el usuario puede pararlos, mientras que si han sido parados previamente (siguiendo el modelo para la gestión de su ciclo de vida definido en la sección 5.4.2.1), el portal permite redesplegarlos o eliminarlos de forma permanente. Asimismo, se pueden consultar los detalles específicos de cada componente de un servicio (estado, nodo en el que han sido desplegados, configuración...).



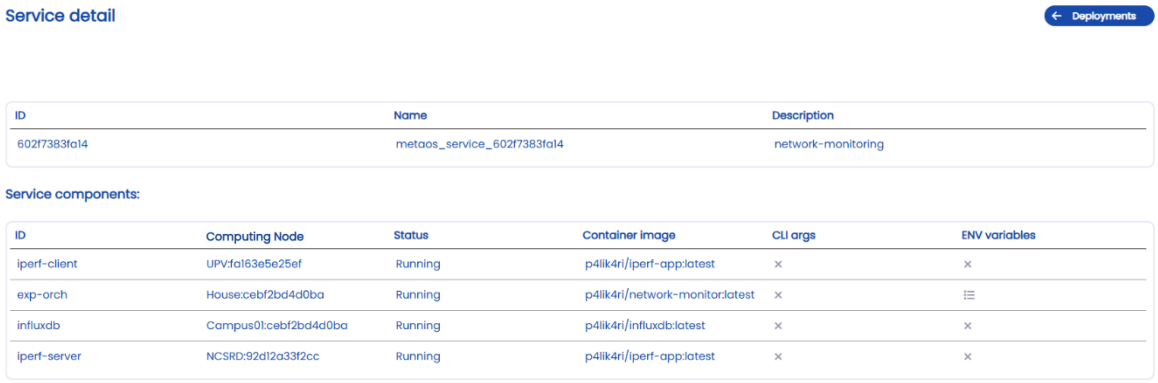
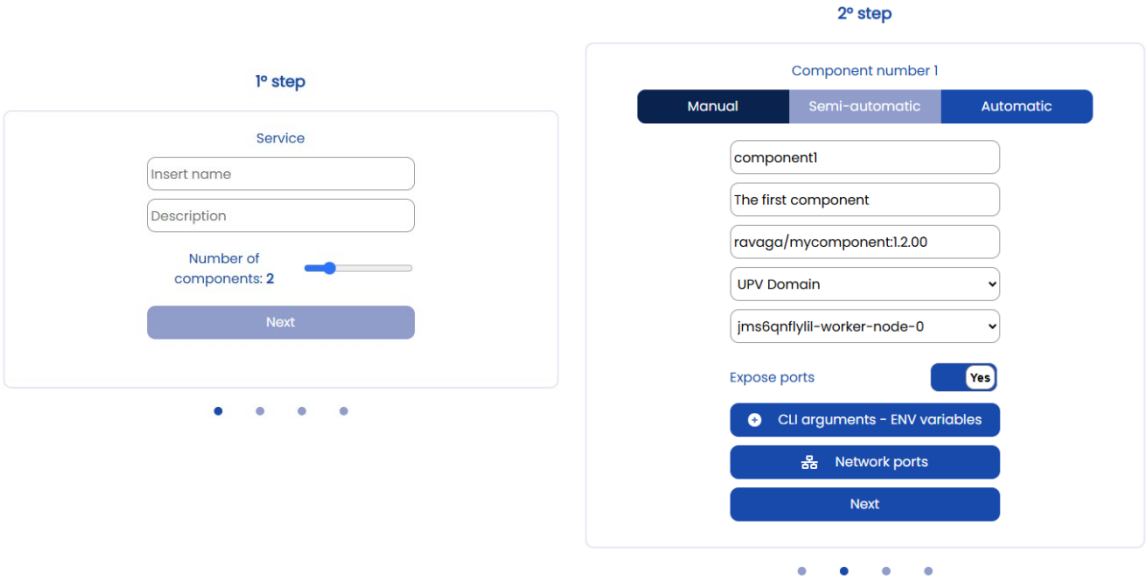


Figura 5.40: Vista de la lista de servicios desplegados y los detalles de sus componentes

Esta vista de servicios no se limita a presentar la información sobre los servicios desplegados, si no que integra una de las partes más importantes del portal de gestión: el despliegue de nuevos servicios en el continuo de computación, actuando como puerta de entrada del sistema de orquestación federado del Meta-OS. Esta funcionalidad se ha implementado en forma de un formulario con múltiples pasos, que comienza con la introducción de las propiedades generales del servicio, continua con el ingreso de las características y configuración de cada uno de los componentes que componen el servicio, y finaliza con un resumen de los parámetros introducidos a lo largo del proceso. Finalmente, cuando el usuario confirma la validez de esta información, el portal envía (a través del Entrypoint balancer) la petición de orquestación al HLO del dominio elegido.



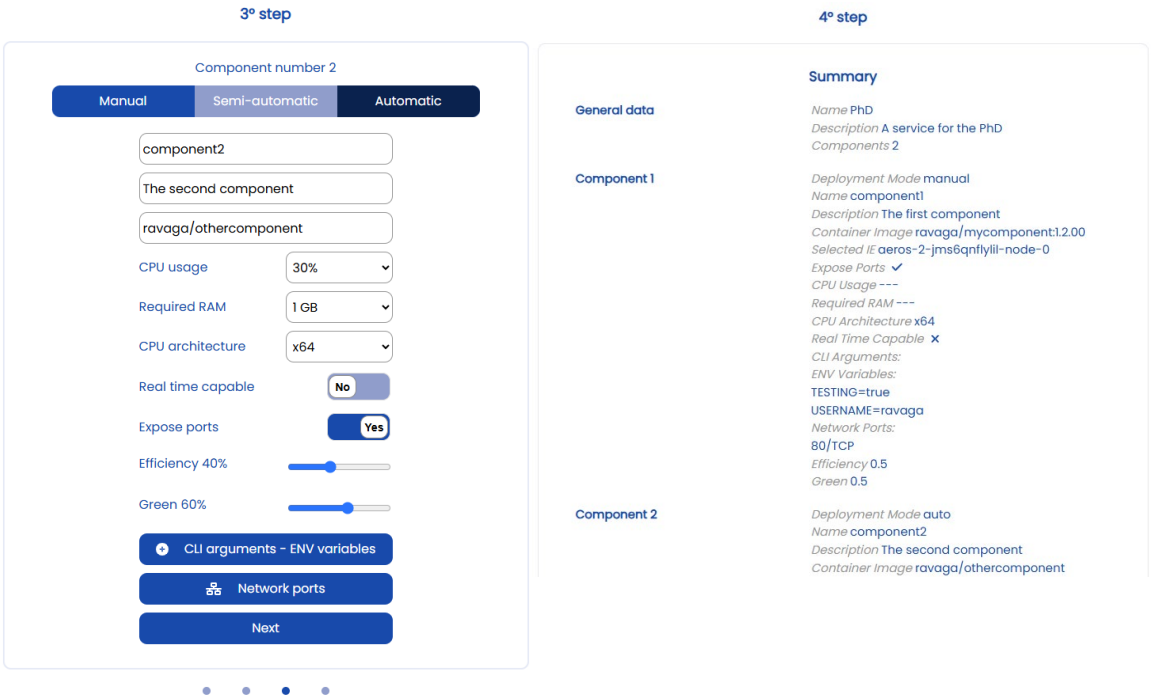


Figura 5.41: Formulario para el despliegue de nuevos servicios en el continuo

Los usuarios que se obtienen directamente de Keycloak también se muestran en una tabla específica, permitiendo su gestión directa sin tener que acceder al panel propio de Keycloak.

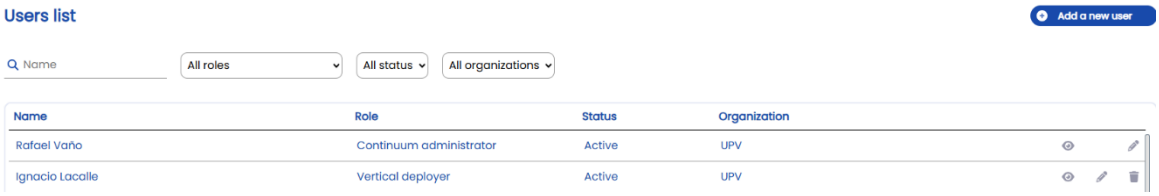


Figura 5.42: Vista de la gestión de usuarios

Por último, las notificaciones se muestran en tiempo real en el escritorio del PC del usuario, siempre y cuando este dote al navegador de los permisos necesarios. Asimismo, se muestra el número de notificaciones que están pendientes de leer, y de manera adicional, el histórico de las mismas se lista en una tabla específica ordenadas de manera descendente por su nivel de prioridad, mediante la cual es posible gestionarlos.

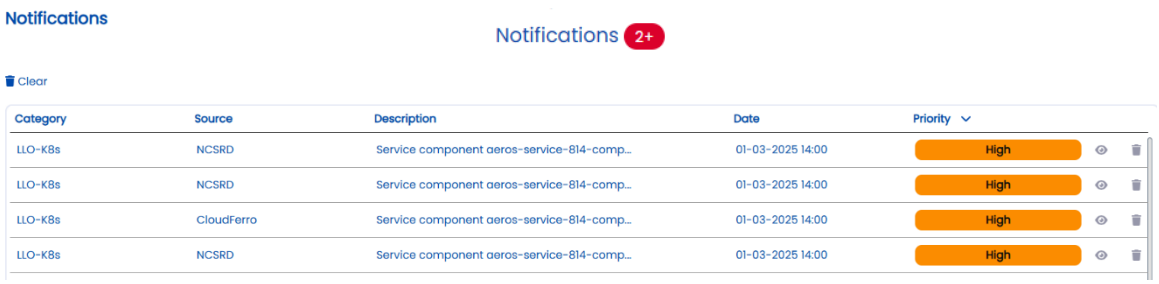


Figura 5.43: Vista de las notificaciones recibidas

5.6. Estrategia de empaquetado y despliegue de la solución

En el momento en el que se define una estrategia de empaquetado común para el Meta-OS que se ha desarrollado durante esta tesis doctoral, ésta ha de aplicar a todos los componentes involucrados en el mismo. En el caso de este Meta-OS, se han identificado dos tipos de componentes sobre los que es absolutamente necesario establecer una serie de diferencias. Estas diferencias se establecen debido al nivel de control que se puede ejercer en cuanto a su empaquetado e identificación para mejorar el nivel de observabilidad global del Meta-OS.

No obstante, ambos tipos de componentes cuentan con una característica común: en última instancia son desplegados utilizando contenedores, ya que esta herramienta de virtualización se ha establecido como la unidad mínima de despliegue del Meta-OS desde un primer momento (ver sección 4.2). El uso de contenedores permite asignar con una serie de etiquetas o *labels* a los contenedores en el momento de su ejecución, con independencia de la estrategia que se ha seguido para crear la imagen de contenedor a partir del código fuente. Por este motivo, todos los contenedores desplegados que formen parte del Meta-OS cuentan con una serie de *labels* comunes a su tipo, como se detallará más adelante.

El primer tipo o grupo de componentes lo constituyen los **servicios desplegados por los usuarios** mediante el componente de orquestación de servicios ofrecido por el Meta-OS. El desarrollo de gran parte de estos componentes es externo al Meta-OS, ya que se puede tratar de cualquier aplicación empaquetada en una imagen de contenedor, por lo que obviamente es imposible controlar su empaquetado, que queda en manos de su equipo de desarrolladores. En este caso, la única opción viable es utilizar una serie de *labels* comunes para identificar de manera inequívoca a los contenedores que corresponden con los componentes que conforman cada servicio de usuario desplegado por el Meta-OS. De esta manera es posible identificarlos hasta el más bajo nivel, es decir, al nivel del sistema de gestión de contenedores que los manejan, además de permitir diferenciarlos de otros contenedores desplegados directamente por usuarios con acceso a dichos sistemas de gestión de contenedores sin que hayan hecho uso del Meta-OS.

Las *labels* comunes para identificar los servicios desplegados por el sistema de orquestación del Meta-OS están basadas en las seis *labels* que recomienda utilizar Kubernetes en su documentación oficial [299]. Sin embargo, para el caso de LLOs diferentes a K8s, se ha reemplazado el prefijo *app.kubernetes.io* por un valor equivalente. Estas *labels* son las siguientes:

- **app.kubernetes.io/name:** nombre del componente de un servicio.
- **app.kubernetes.io/instance:** instancia única de un componente, que sigue el formato *metaos-service-{serviceID}-component-{componentName}*.

- **app.kubernetes.io/part-of:** servicio al que pertenece el componente con el formato *meta-os-service-{serviceID}*.
- **app.kubernetes.io/managed-by:** siempre tiene el valor *meta-os.eu*, para indicar que está gestionado por este Meta-OS.
- **app.kubernetes.io/created-by:** identificador del LLO que lo ha desplegado.

A nivel operacional, estas *labels* son introducidas por el LLO correspondiente. Dependiendo del tipo de LLO, las *labels* son únicamente creadas en el mismo contenedor en el caso de Docker o containerd, o en todos los artefactos creados por el LLO en el caso de K8s (*Deployments, Services, ConfigMaps, Secret*, etc.). Esto es debido a que en K8s el uso de las *labels* está ampliamente establecido como parte de su modelo descriptivo para definir y gestionar recursos.

Por otra parte, los **componentes o módulos que realmente constituyen los bloques que habilitan el Meta-OS** han sido desarrollados desde cero por parte del doctorando, lo que significa que el proceso de empaquetado de los mismos ha estado bajo su control en todo momento. Por ende, la acción más lógica consiste en la definición de una estrategia de empaquetado común para la instalación de los bloques que conforman el Meta-OS, que además sea de utilidad para su identificación, organización y gestión posteriormente a su despliegue.

Llegados a este punto se va a realizar una pequeña aclaración para no confundir al lector con los términos empleados. El bloque de orquestación está compuesto por diferentes módulos (HLO y LLO), que a su vez están compuestos por múltiples módulos *software*. El LLO K8s está solamente formato por el Operador, pero el módulo LLO Docker está formado por varios componentes: Operador, controlador Docker y *broker* NATS. El término "componente" dentro de, por ejemplo, un Helm chart, se utiliza para identificar un artefacto de despliegue de K8s como un *Deployment* o un *StatefulSet*, mientras que en un archivo Docker Compose, se utilizaría para identificar un *service* (un contenedor junto con sus parámetros de despliegue específicos).

Siguiendo con el desarrollo de la sección, la gran diferencia con la que cuentan los bloques esenciales del Meta-OS es que deben ser instalados manualmente sobre infraestructura existente (clústeres de K8s, VMs con Docker instalado...), sin contar con un elemento de abstracción de la infraestructura subyacente como lo es el sistema de orquestación proporcionado por el Meta-OS. Por lo tanto, debe escogerse para qué plataformas va a estar disponible la instalación del Meta-OS, y empaquetar sus módulos con herramientas acordes a dicha decisión. En este punto conviene aclarar que esto no significa que el Meta-OS no contemple el despliegue de servicios (en forma de contenedores) en otro tipo de infraestructura, ya que era un requisito técnico (plasmado en la Tabla 4.1), cuyo cumplimiento se ha descrito con detalle en el apartado previo 5.4. Además, los bloques básicos del Meta-OS han de ser instalados en los nodos manualmente (es decir, por los administradores de los dominios, siguiendo su propio criterio en cuanto a la instalación y distribución

de los mismos, puesto que no van a ser desplegados, como es lógico, utilizando el sistema de orquestación de servicios del Meta-OS, al tratarse de dos tipos de despliegue diferentes).

Excepto los Operadores de K8s de los LLOs que solamente pueden ser instalados en clústeres de K8s, requiriendo la existencia de un clúster mínimo en cada dominio para este propósito (incluso puede ser instalado utilizando alguna de las distribuciones ligeras de K8s descritas en la sección 2.6.5.2), el doctorando ha decido que la instalación del Meta-OS solamente esté disponible para las siguientes plataformas:

- **Máquinas independientes con Docker Engine** instalado: los diferentes bloques y módulos del Meta-OS están disponibles para su instalación en archivos Docker Compose. Aunque se recomienda el uso de K8s para la instalación de los componentes claves debido a que es el estándar *cloud-native* para la orquestación de contenedores, aún existe una gran cantidad de infraestructura en entornos industriales que solamente cuenta con Docker para gestionar la ejecución de contenedores, como se ha podido comprobar en los casos de uso descritos en el capítulo 6. Incluso en máquinas con solamente containerd instalado se podría hacer uso de la herramienta contaiNERD CTL para ejecutar estos archivos Docker Compose.
- **Clústeres de K8s**: es obligatorio que la instalación del Meta-OS esté disponible para la herramienta estándar de orquestación de contenedores (K8s) mediante una herramienta de empaquetado *cloud-native* ampliamente establecida y que permita una gran versatilidad en cuanto a la configuración de los despliegues para que los mismo sean lo más directos posible, como es Helm. Por lo tanto, en este caso la instalación se realiza exclusivamente mediante Helm charts, descartando el uso directo de manifiestos de K8s u otras herramientas como Kustomize.

La creación de una serie de *labels* comunes para identificar de manera inequívoca a los módulos elementales del Meta-OS también es necesaria, además de facilitar su monitorización utilizando herramientas *cloud-native* como Prometheus. La gran diferencia es que ha de realizarse manualmente sobre los artefactos en los que se empaquetan, es decir, sobre los archivos Docker Compose o sobre los Helm charts. En esta acción queda patente la ventaja de utilizar una herramienta basada en *templating* como Helm, dado que permite la creación automática de contenido (como, por ejemplo, un identificador único de la instancia desplegada), que también puede utilizar como fuente ciertos valores introducidos por los usuarios.

Sin más dilación, las *labels* que se han definido son las siguientes (en el caso de Docker se ha reemplazado el prefijo *app.kubernetes.io/* por *app.docker.io.*):

- **app.kubernetes.io/name:** nombre personalizado del módulo. En el caso de Helm corresponde con el valor de la variable *application.name*

definida en el archivo *_helpers.tpl*. Esta variable tiene como valor el nombre del *chart* o el de *nameOverride* si ha sido modificado.

- **app.kubernetes.io/instance:** instancia única de un módulo. En Helm corresponde con la versión del *release* del chart.
- **app.kubernetes.io/version:** versión del módulo. En Helm es la *appVersion* del chart.
- **app.kubernetes.io/component:** el nombre del componente específico del módulo.
- **app.kubernetes.io/managed-by:** la herramienta utilizada para desplegar y gestionar el módulo (valores *Helm* o *Docker-compose*).
- **tier:** por defecto, se asigna el valor *external* para el componente principal e *internal* para los demás. Sin embargo, puede modificarse para diferenciar entre los componentes del mismo (por ejemplo, *api* y *database*).
- **isMainInterface**: valor booleano que indica si el componente es la interfaz principal. El primer componente se considera la interfaz principal y puede utilizarse, por ejemplo, como punto de partida para la creación de *Network Policies* en K8s.
- **helm.sh/chart:** (solamente para Helm charts) el valor de la variable *application.chart* definida en el archivo *_helpers.tpl*. Se trata de una cadena de texto compuesta por el nombre del chart y su versión.

5.6.1. Helm chart generator

La creación de un Helm chart desde cero para empaquetar un desarrollo *software* no se trata de una acción trivial, puesto que no solo requiere un conocimiento previo de la estructura de los *charts* y de su funcionamiento, sino también del despliegue de aplicaciones en K8s mediante la creación de manifiestos. Además, en el caso del desarrollo de los módulos elementales de este Meta-OS, los *charts* creados para empaquetarlos han de seguir la estrategia de empaquetado descrita en los párrafos previos.

El doctorando fue percatándose a partir de su participación en el proyecto ASSIST-IoT (donde se empezaron a utilizar Helm charts para empaquetar y desplegar los desarrollos realizados en K8s, ver 3.3.1) que la mayoría de los desarrolladores que contaban con una experiencia limitada en K8s, o que directamente usaban manifiestos de K8s para desplegar aplicaciones, tenían dificultades para crear *charts* que realmente aprovecharan todo el potencial de esta tecnología. Esto se traducía en que a menudo estos *charts* simplemente agrupaban manifiestos de K8s (ficheros YAML) poco configurables o extensibles, alejándose del espíritu con el que se desarrolló Helm. Por este motivo, el doctorando empezó el desarrollo de una herramienta tipo CLI para la generación automática de Helm

charts a partir de los parámetros introducidos por los usuarios en la línea de comandos, con una estructura previamente definida basada en las convenciones oficiales y en las buenas prácticas recomendadas por la documentación oficial de Helm [300].

Esta herramienta, llamada Helm chart generator, ha sido desarrollada como una aplicación *stand-alone* utilizando Node.js, utilizando principalmente las librerías *handlebars* (compilación de *templates*), *commander* e *inquirer*. El funcionamiento de esta herramienta es simple: en primer lugar, se pide al usuario que introduzca en la línea de comandos una serie de características generales sobre la aplicación de la que se desea crear el Helm chart en cuestión (nombre, versión, número de componentes, *(Cron)Jobs* y dependencias de otros charts). Seguidamente, para cada componente de la aplicación, se requiere la información específica del mismo (imagen, tipo —*Deployment, StatefulSet* o *DaemonSet*—, puertos, variables de entorno...), repitiéndose el mismo proceso para los *Jobs, CronJobs*, y dependencias de otros *charts*. La totalidad de las preguntas realizadas o información requerida se puede consultar en su documentación oficial [301].

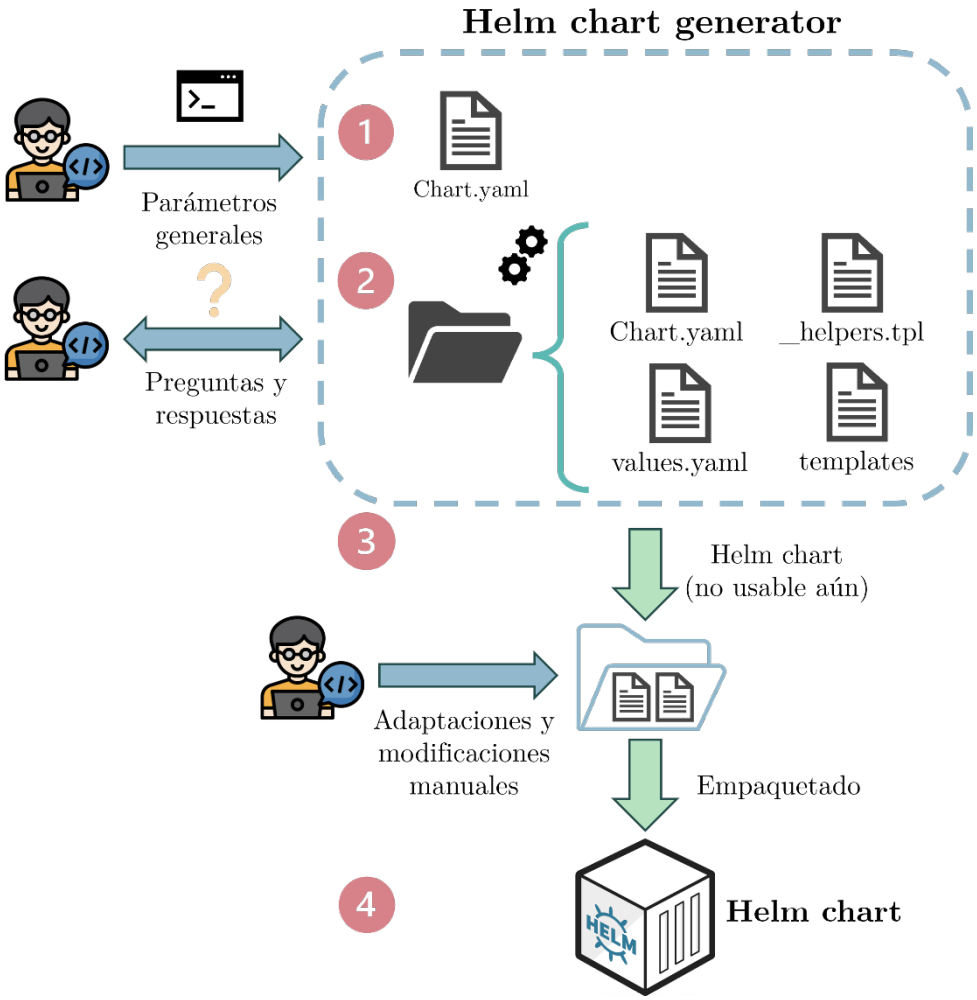


Figura 5.44: Funcionamiento del Helm chart generator

Esta herramienta cuenta con una serie de *templates* predefinidos que están basados en los *templates* y la estructura de *values* definidos por Bitnami, bajo licencia de código abierto [302]. El Helm chart generator procede a rellenar estos *templates* predefinidos con la información introducida por el usuario, para finalmente crear los archivos necesarios (carpetas, manifiestos en YAML, *subcharts*, etc.). Además, incluye por defecto las *labels* definidas en la estrategia de empaquetado de esta tesis en todos los componentes de K8s, y cuyo valor es rellenado de manera dinámica, facilitando el empaquetado de los componentes esenciales del Meta-OS.

Por último, se crea un Helm chart con la estructura de carpetas definida y un archivo *values* totalmente adaptado para la aplicación. Sin embargo, este *chart* aún no está listo para ser usado, puesto que necesita la revisión manual del desarrollador para realizar ciertas modificaciones, como variables de entorno o configuraciones adicionales o específicas no soportadas por esta herramienta. Después de esta revisión, este Helm chart puede ser directamente utilizado en un clúster de K8s, o ser empaquetado mediante el comando *helm package* para ser subido a un repositorio de Helm charts, como, por ejemplo, Artifact Hub, que se trata del mayor repositorio abierto de Helm charts.

Lógicamente, esta herramienta ha sido utilizada para crear todos los Helm charts de los desarrollos realizados durante esta tesis. Además, ha sido publicada en un repositorio de GitHub bajo una licencia de código abierto, en el que se pueden consultar todos los detalles de funcionamiento, configuración o estructura del chart [301]. Asimismo, su imagen de contenedor está disponible en Docker Hub. El Helm chart generator sirve como demostración del espíritu global de los desarrollos realizados en esta tesis, ya que puede utilizarse para empaquetar todo tipo de desarrollos para ser desplegados en K8s, sin que tengan una relación directa con el continuo de computación. Su objetivo desde un primer momento ha sido ejercer como herramienta de apoyo para mejorar la adopción de soluciones *cloud-native*.

Finalmente, en la siguiente tabla se muestra de manera detallada la estructura de los Helm charts generados, junto con una breve descripción de cada elemento generado.

Tabla 5.2: Estructura del Helm chart generado por la herramienta Helm chart generator

nombre-de-la-aplicación/	
Chart.yaml	Fichero YAML que contiene información general sobre el *chart*.
.helmignore	El archivo *.helmignore* se utiliza para especificar los ficheros que no se deben incluir en el *chart*.
values.yaml	Los valores de configuración por defecto del *chart*. Estos valores están agrupados por el componente o el *(Cron)Job* al que pertenecen, dentro de un objeto diferente.
qa-values.yaml	Una copia del *values.yaml* para propósitos de desarrollo.

charts/	Una carpeta que contiene cualquier chart (*subchart*) del que depende este *chart.* Esta carpeta está inicialmente vacía, por lo que el usuario debe incluir dentro de ella los *subcharts* necesarios, o añadirlos automáticamente usando el comando *helm dependency update.*
crds/	*Custom Resource Definitions* (carpeta vacía).
templates/	Un directorio de *templates* que, cuando se combinan con los *values*, generan ficheros con manifiestos de K8s válidos.
NOTES.txt	Un fichero de texto en plano que contine notas breves sobre el uso del chart. Se genera un archivo por defecto.
_helpers.tpl	En este fichero se definen el nombre del módulo o aplicación que se desea empaquetar, el nombre del *chart*, el nombre y *labels* de los componentes (que en este caso son las definidas en la estrategia de empaquetado), así como de los *(Cron)Jobs.*
componenteN	Este directorio contiene los manifiestos que definen el despliegue de los artefactos de K8s (*Deployment, StatefulSet* o *DaemonSet*), junto con los manifiestos de sus K8s *Services*, del componente N de la aplicación. El usuario puede incluir manualmente dentro de esta carpeta manifiestos de K8s adicionales (por ejemplo, *ConfigMaps*, *PVCs*, *Secrets*). **Nota:** se crea una carpeta para cada componente del *chart.*
jobs/	Este directorio contiene todos manifiestos de K8s relacionados con los *Jobs* y *CronJobs.* Solamente se crea si la aplicación contiene *Jobs* o *CronJobs.*

5.7.Elementos adicionales

El capítulo 5 ha estado completamente enfocado en la implementación de la solución propuesta por esta tesis, es decir, los componentes claves que habilitan la creación de un Meta-OS para la gobernanza del continuo *Cloud-Edge-IoT*. Sin embargo, para que la implementación de un Meta-OS fuera completa, debería contar con otros elementos transversales y de gran importancia para el mismo.

Uno de los elementos transversales más importantes se trata de la seguridad. La decisión para no calificar la seguridad como un bloque básico del Meta-OS radica en que este aspecto o área de conocimiento constituye una temática de investigación por sí misma, la cual es más desafiante si se enmarca en el contexto del continuo de computación o de entornos altamente distribuidos. Sin ir más lejos, la principal línea de investigación sobre la que pivota la tesis doctoral de algunos compañeros del grupo de investigación SATRD se trata del diseño e implementación de sistemas de seguridad altamente distribuidos en el contexto del continuo CEI. Por lo tanto,

de haber puesto el foco sobre esta temática, el doctorando podría haberse desviado de su línea de investigación principal.

Aunque durante esta tesis no se hecho demasiado hincapié en la seguridad con respecto al continuo o al propio Meta-OS, al no haber sido identificada como uno de los bloques básicos que habilitan la creación y el funcionamiento de un Meta-OS, el doctorando quiere destacar que este aspecto ha estado presente durante el diseño de dichos bloques fundamentales, así como en la elección de las tecnologías para desarrollarlo. Un claro ejemplo de ello es el acceso al portal de gestión y la asignación de roles a los diferentes usuarios para que solamente puedan realizar las acciones específicamente asignadas a estos. Además, el Meta-OS está preparado para proteger el acceso de la API REST de Orion-LD con el despliegue de una API *Gateway* o *proxy* mediante la comprobación de *tokens* incluidos en las peticiones de acceso.

Por otra parte, las comunicaciones entre dominios enmarcadas dentro de la federación están encriptadas y son completamente seguras, ya que la librería *libp2p* provee estas funcionalidades de manera intrínseca, encriptando todos los datos que se envían a través de sus *streams*. Asimismo, las conexiones entre componentes se realizan a través de túneles de WireGuard, por lo que las comunicaciones están cifradas extremo a extremo. En la misma línea, para los dominios públicos está estrictamente recomendado el uso de certificados y el uso del protocolo TLS para cifrar y proteger las conexiones, especialmente para el acceso a la plataforma de gestión del Meta-OS.

De manera adicional, a los diferentes Operadores de los LLOs desarrollados se les dotó con una integración con el sistema de fiabilidad del Meta-OS de aerOS, que está basado en IOTA, una tecnología de registro de datos distribuida (DLT, del inglés Distributed Ledger Technology) de código abierto, que fue diseñada específicamente para su uso en el ámbito del IoT [303]. A diferencia de las cadenas de bloques tradicionales, IOTA utiliza una estructura llamada *tangle*, que se trata de un Grafo Acíclico Dirigido (DAG). Mediante esta integración, se habilita la creación de un registro inmutable de los eventos clave que han tenido lugar durante el proceso de orquestación de un servicio, dotando al Meta-OS de un mayor de observabilidad que a su vez aumenta el nivel de confianza en el mismo [304]. Este registro podría extenderse a los demás elementos básicos del Meta-OS, así como a su sistema de alertas. En la parte final del apartado 6.3 se podrá validar la implementación de esta funcionalidad adicional.

Capítulo 6

Casos de uso

6.1.Introducción

Para demostrar la viabilidad del sistema desarrollado en esta tesis, es decir, los componentes fundamentales que habilitan un Meta-OS para gobernar el continuo de computación *Cloud-Edge-IoT*, es fundamental ponerlo a prueba en escenarios reales que permitan evaluar su funcionamiento y aplicabilidad en diferentes entornos, así como confirmar su aceptación por parte de los usuarios a los que se dirige. En el ámbito de esta tesis doctoral, estos entornos corresponden con diferentes continuos de computación compuestos por una gran variedad de infraestructura, puesto que el continuo ya es heterogéneo por definición. En términos generales, esta sección tiene como objetivo evidenciar la fase de experimentación de la tesis doctoral.

Tomando este objetivo global como punto de partida se han seleccionado tres casos de uso, cada uno con un nivel de madurez tecnológica progresivo según la escala *Technology Readiness Level* (TRL) [305]. Esta escala, desarrollada por la NASA en la década de los años 70, evalúa la madurez de una tecnología,

comprendiendo desde la investigación básica hasta su implementación en entornos operativos. Además, desde 2014 ha sido adoptada por la Comisión Europea para la evaluación del alcance de las soluciones desarrolladas dentro de las iniciativas de investigación pertenecientes a los programas Horizon 2020 y Horizon Europe, entre otros.

La selección de estos casos de uso responde a una estrategia de validación escalonada, donde cada caso de uso introduce nuevos desafíos y requisitos que permiten evaluar la madurez tecnológica del Meta-OS en escenarios de creciente complejidad. En resumen, los tres casos de uso abordados en este capítulo siguen una progresión ascendente en términos de TRL, empezando por el nivel 4 hasta acabar con el nivel 6. Los casos de uso correspondientes a los niveles 5 y 6 de TRL han sido posibles gracias a la participación del doctorando en proyectos de investigación europeos por formar parte como investigador del grupo de investigación SATRD de la UPV (como se apuntó en el capítulo 3), especialmente en el proyecto aerOS. Objetivamente, el doctorando quiere destacar que sin la participación en este tipo de proyectos la validación del sistema desarrollado en esta tesis habría tenido un mayor grado de dificultad, puesto que desde la propia universidad es difícil tener acceso directo a entornos relevantes relacionados con la industria. Este hecho puede ser visto como un argumento a favor para la existencia de proyectos de investigación colaborativos, en los que se fomenta la validación de los desarrollos realizados en entidades de investigación en agentes relevantes de la industria europea.

En los siguientes apartados, se va a proceder a detallar los tres casos de uso seleccionados, explicando sus objetivos, requisitos y los aprendizajes obtenidos en cada uno de ellos. Pero antes de saltar a dichos apartados, es relevante realizar una pequeña introducción de los mismos:

- **Laboratorio:** en este caso de uso se va a utilizar el laboratorio del grupo de investigación SATRD para desplegar un continuo de computación simulado en un entorno controlado. El objetivo principal de este caso de uso es la validación del Meta-OS desarrollado en un entorno con un gran nivel de heterogeneidad, demostrando su aplicabilidad en una gran variedad de dispositivos. Corresponde con un nivel 4 de TRL.
- **Prueba relevante en un entorno controlado:** este escenario corresponde con un despliegue prototípico y paradigmático para entornos IoT, que fue utilizado como base para la defensa del resultado tecnológico del proyecto aerOS frente a la Comisión Europea. Corresponde con un nivel 5 de TRL.
- **Despliegues operativos reales:** este caso de uso, comprendido a su vez por tres entornos operativos reales, pretende demostrar que el despliegue del Meta-OS ha actuado como el elemento habilitador clave para la consecución de sus objetivos finales. Corresponde con un nivel 6 de TRL.

6.2.Laboratorio del grupo SATRD

El primer escenario de validación consiste en la configuración y despliegue de un continuo de computación simulado, diseñado bajo el criterio del doctorando, en el laboratorio del grupo de investigación SATRD de la UPV. Este escenario ha sido posible gracias a la potente infraestructura, tanto *cloud* como *edge,* con la que cuenta el grupo SATRD [306], permitiendo la creación de tantas máquinas virtuales como fueran requeridas, así como el uso de diversos SBCs y dispositivos IoT (sensores y actuadores). Resulta evidente que este escenario debe ser clasificado como un nivel 4 de TRL ("validación de la tecnología en el laboratorio"), ya que se va a integrar y se realizan pruebas con el Meta-OS en un entorno totalmente controlado, que ha sido completamente diseñado e implementado por el mismo doctorando para que encaje perfectamente con el supuesto que se pretende validar. Además, parte de la infraestructura de este caso de uso ha sido utilizada durante el proceso de desarrollo del *software* de los componentes esenciales del Meta-OS, permitiendo agilizar este proceso al disponer de infraestructura en la que realizar pruebas de implementación y depuración de la tecnología.

El principal propósito que persigue este escenario de validación es la creación de un continuo de computación con el mayor grado de heterogeneidad posible para validar la instalación y el funcionamiento de los bloques fundamentales del Meta-OS desarrollados en dispositivos de diferente naturaleza. Es necesario reconocer que para alcanzar tal nivel de heterogeneidad se ha creado un continuo que podría considerarse como artificial, es decir, con una composición que no es esperable encontrar en escenarios de aplicación reales. Esta heterogeneidad se ha intentado plasmar en los diferentes niveles que se enumeran a continuación, y queda reflejada en las distintas configuraciones aplicadas a los diversos nodos desplegados para la constitución del continuo de computación enumerados en la Tabla 6.1. Adicionalmente, con el fin de ilustrar de una manera más visual esta diversidad, en la Figura 6.2 se muestran algunas imágenes reales de dichos nodos:

1. **Tipo de nodo o máquina**: máquinas virtuales, máquinas físicas, dispositivos *edge* industriales, Raspberry Pis, …
2. **Sistema operativo**: Talos Linux, Ubuntu, Raspberry Pi OS…
3. **Arquitectura de CPU**: AMD64, ARM64 y RISC-V.
4. **Capacidad de computación**: núcleos y potencia del procesador y cantidad de memoria RAM.
5. **Tipo de red a la que está conectado** (LANs, redes internas de la universidad, IPs públicas de Internet...) y **medio de conexión a la red** (cable e inalámbrico).
6. **Sistema de gestión de contenedores**: múltiples distribuciones de K8s (K8s puro o *upstream*, K3s y KubeEdge), Docker, containerd y Balena.

7. ***Runtime* de contenedores de alto nivel** (containerd y CRI-O) y **de bajo nivel** (runc, crun y youki, descritos en el apartado 2.6.3).

Se ha incidido durante esta tesis (ver sección 4.2) en que una de las primeras tareas a la hora de desplegar un continuo de computación es decidir la estructura y topología de los dominios, es decir, la agrupación de nodos de computación según determinados criterios. En este sentido, en el primer caso de uso, se ha decidido agrupar los nodos de computación en tres dominios diferentes, tal y como queda reflejado en la Figura 6.1, así como en la Tabla 6.1. Es necesario recordar que cada dominio ha de contar con su propia instancia de ciertos elementos claves del Meta-OS (HLO, LLOs, gestor de contexto...). De este modo, se han desplegado los componentes desarrollados pertenecientes a los bloques fundamentales del Meta-OS en todo el continuo, validando que los diferentes componentes de monitorización alimentaran a los CBs de todos los dominios y que esta información estuviera accesible de manera ubicua gracias a la federación de los dominios. Además, se realizaron las pertinentes pruebas de orquestación de servicios para validar el correcto funcionamiento de todos los tipos de LLOs desarrollados.

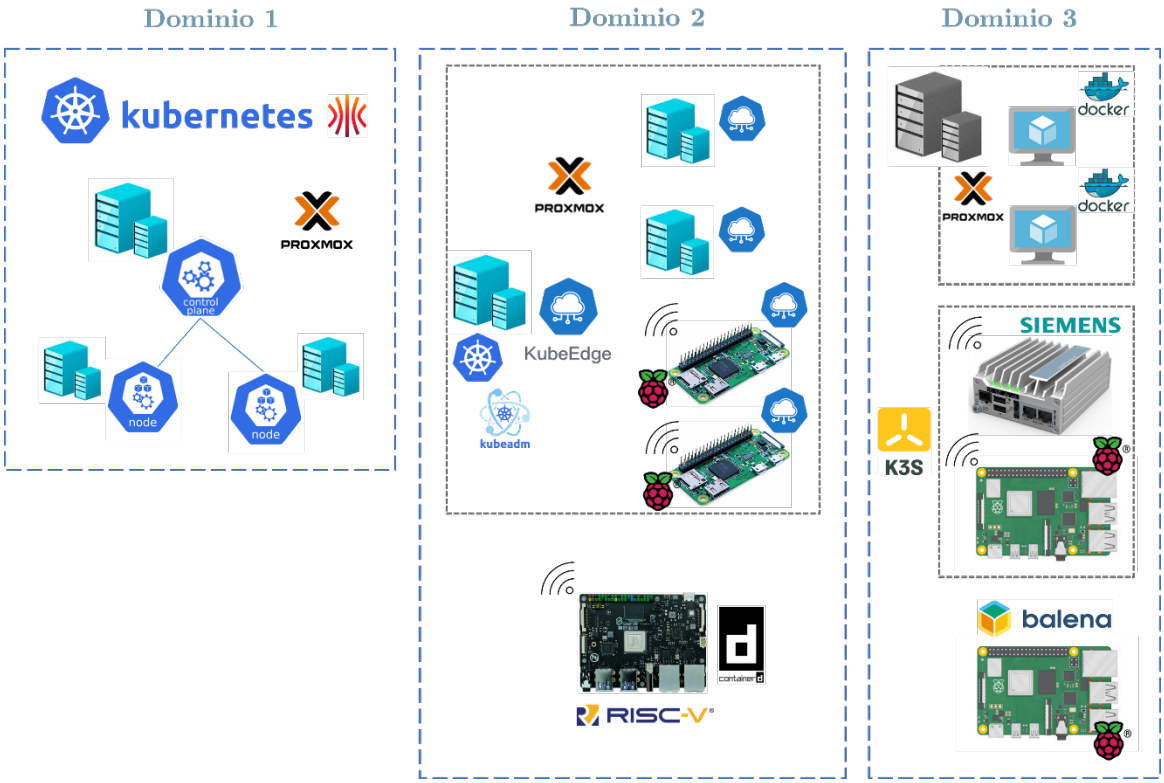


Figura 6.1: Continuo del laboratorio del grupo de investigación SATRD

El primer dominio, seleccionado como dominio *entrypoint*, está formado por un clúster de K8s constituido usando Talos Linux [307] (un OS de código abierto específicamente diseñado para K8s y certificado por la CNCF) de tres nodos (un nodo del plano de control más dos nodos *workers*), que han sido creados como máquinas virtuales utilizando la herramienta de virtualización Proxmox. Este

dominio cuenta con una IP pública, por lo que puede ser accedido desde fuera de la red interna de la UPV, y sería equivalente a un dominio puramente *cloud*.

El segundo dominio está principalmente formado por un clúster de KubeEdge (ver 2.6.5.2) para demostrar la integración del sistema de gestión de contenedores *Cloud-Edge* oficial (*de facto*) de la CNCF. El nodo *cloud* de este clúster cuenta con una IP pública a nivel de la red interna de la universidad, permitiendo la conexión a este clúster de nodos *edge* que estén conectados a cualquier red interna de la universidad. Este nodo principal ha sido creado utilizado el mismo Proxmox que en el anterior clúster, al igual que dos de los nodos *edge* del clúster. Los dos nodos restantes del clúster de KubeEdge se tratan de dos Raspberry Pi Zero 2W, que solamente cuentan con 512 MB de RAM, demostrando que incluso dispositivos con capacidades tan limitadas de computación pueden formar parte del continuo y ser gestionadas por el Meta-OS desarrollado. Asimismo, el uso de un clúster KubeEdge, que cuenta con su propia arquitectura *Cloud-Edge*, puede ser visto como una prueba de la gran flexibilidad con la que cuenta el Meta-OS. Por último, este dominio cuenta con tipo de SBC llamado StarFive Vision Five 2 [308], una de los pocos SBCs comerciales que utilizan una CPU con arquitectura RISC-V que son accesibles para el público en general. El propósito de incluir este dispositivo es la validación del LLO containerd, así como validar el primer paso hacia la inclusión en el continuo (bajo la gestión del Meta-OS) de dispositivos cuyo diseño *hardware* se ha realizado bajo licencias de código abierto.

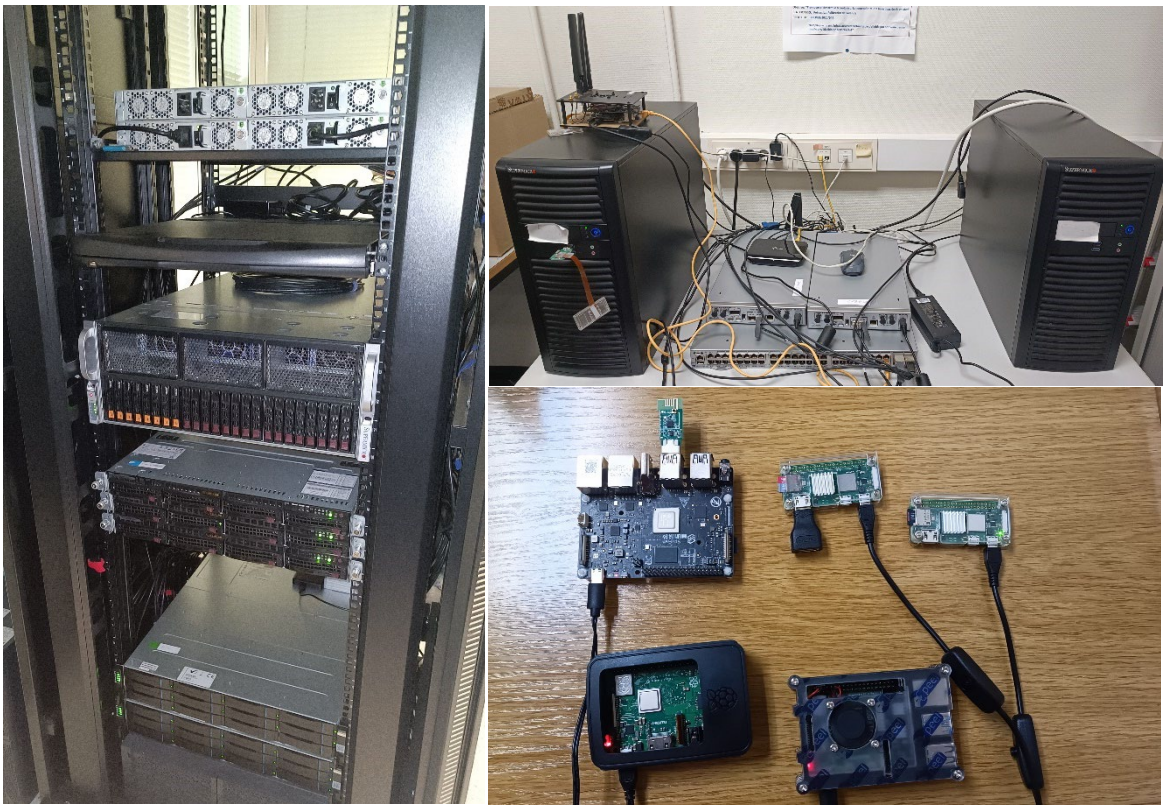


Figura 6.2: Imágenes reales de la infraestructura del laboratorio SATRD

El tercer dominio es uno de los más heterogéneos en cuanto a sistemas de gestión de contenedores. Contiene dos VMs, que utilizan Docker como su sistema de gestión de contenedores, con suficientes capacidades de computación para instalar los elementos claves del Meta-OS, demostrando que estos elementos pueden ser instalados tanto en K8s (caso de los dominios 1 y 2) como en Docker, tal y como está contemplado en la estrategia de despliegue definida en la sección 5.6. Asimismo, este dominio cuenta con un clúster de K3s (una distribución ligera de K8s) con capacidades de computación más reducidas, en el que se integra un dispositivo industrial *edge*, que es utilizado para ejecutar los Operadores de los diferentes tipos de LLOs del dominio (K8s, Docker y Balena). Finalmente, se decidió añadir una Raspberry Pi con el balenaOS instalado para validar este tipo de LLO.

Tabla 6.1: Nodos de computación que forman parte del continuo del laboratorio del grupo de investigación SATRD

Nodo	OS	Arquitectura CPU	RAM	Sistema de gestión de contenedores
VM *control plane* clúster	Talos Linux 1.9.2	AMD64	32 GB	Kubernetes 1.28 (containerd 1.17.25)
VM *worker* 1 clúster	Talos Linux 1.9.2	AMD64	16 GB	Kubernetes 1.28 (containerd 1.17.25)
VM *worker* 2 clúster	Talos Linux 1.9.2	AMD64	16 GB	Kubernetes 1.28 (CRI-O 1.28.2)
VM *control plane* clúster	Ubuntu server 22.04.3	AMD64	16 GB	Kubernetes 1.29 (containerd 1.17.25)
VM nodo *edge* 1	Ubuntu server 22.04.3	AMD64	4 GB	KubeEdge 1.19.1 (containerd 1.17.25)
VM nodo *edge* 2	Ubuntu server 22.04.3	AMD64	2 GB	KubeEdge 1.19.1 (CRI-O 1.29)
Raspberry Pi Zero 2W 1	Raspberry Pi OS 12	ARM64	512 MB	KubeEdge 1.19.1 (containerd 1.17.25)
Raspberry Pi Zero 2W 2	Raspberry Pi OS 12	ARM64	512 MB	KubeEdge 1.19.1 (containerd 1.17.25)
Vision Five 2 RISC-V	Versión adaptada de Debian 12	ARM64	4 GB	containerd 1.17.25
VM Docker 1	Ubuntu server 22.04.3	AMD64	8 GB	Docker 28.0.1
VM Docker 2	Ubuntu server 22.04.3	AMD64	8 GB	Docker 28.0.1

Raspberry Pi 4	Ubuntu server 22.04.3	ARM64	4 GB	K3s 1.28.3 (containerd 1.7.11)
SIMATIC IPC127E	SIMATIC Industrial OS	AMD64	4 GB	K3s 1.28.3 (containerd 1.7.11)
Raspberry Pi 5	Balena OS 6.3.23	ARM64	8 GB	balenaEngine 20.10.43

Dejando a un lado las características de la infraestructura que compone el continuo de computación de este caso de uso, el doctorando ideó un flujo de validación práctico para ser ejecutado en este continuo. Este flujo comienza con la adición de un nuevo dominio al continuo, pero en este caso localizado físicamente en la propia casa del doctorando, significando la entrada en escena de una red local privada en el continuo, ya que hasta este momento todos los nodos estaban conectados a la red interna de la UPV. Este dominio está compuesto por una Raspberry Pi junto con un ordenador físico, formando ambos un clúster de K3s. En esta infraestructura se alojan los servicios necesarios para habilitar el funcionamiento del entorno *smart home* del doctorando, que está construido entorno a la herramienta de automatización del hogar Home Assistant (HA) [309], junto con una serie de dispositivos IoT domóticos conectados vía el protocolo ZigBee.

La adición de este nuevo dominio, y la consecuente federación del mismo con los dominios existentes, fue posible gracias a la instalación y ejecución del componente Federador (explicado en el punto 5.3.1.1), en combinación con los mecanismos implementados para lograr el establecimiento de comunicaciones entre dominios pertenecientes a diferentes tipos de redes detallado en el apartado 5.3.2. En este escenario, fue necesario abrir los puertos necesarios al exterior en el cortafuegos corporativo de la entidad administrativa en la que está situado el laboratorio del SATRD, para habilitar a que dominio 1 actuara como *relay* para el establecimiento de los canales de comunicaciones necesarios entre dominios.

Una vez establecida la federación entre todos los dominios, se instaló un sensor de presencia en el puesto de trabajo del doctorando para que detectara su presencia en el mismo. De forma paralela, en la casa inteligente del doctorando se encuentran instaladas una serie de bombillas inteligentes. El escenario diseñado consiste en la implementación de un proceso de automatización en el que, cuando el sensor de presencia detecte que el doctorando ha acudido a su puesto de trabajo, automáticamente se compruebe que en su casa no haya ninguna bombilla inteligente encendida, y en caso contrario, que se apague para ahorrar energía.

Este escenario de automatización es fácilmente realizable gracias al Meta-OS desarrollado. En primer lugar, se desarrolló un pequeño agente NGSI-LD para crear y actualizar una entidad NGSI-LD que reflejara el sensor de posición, que además realiza un preprocesamiento de los datos recibidos del sensor para evitar falsos positivos. Este agente se desplegó de manera manual (modo manual de

orquestación) en una de las Pi Zero del dominio 2, situada al lado del sensor (fuente de datos), para que la latencia sea mínima. Una vez la entidad estuvo presente en el CB del dominio 2, el agente crea una suscripción de manera federada (ver 5.3.1) en el CB del dominio de la casa para que envíe una notificación al *endpoint* de Home Assistant relacionado con el *webhook trigger* [310], que procesa reglas de automatización a partir de la recepción de una petición HTTP POST. Por último, se creó la regla de automatización en HA para que apagara las luces cuando reciba la notificación del CB.

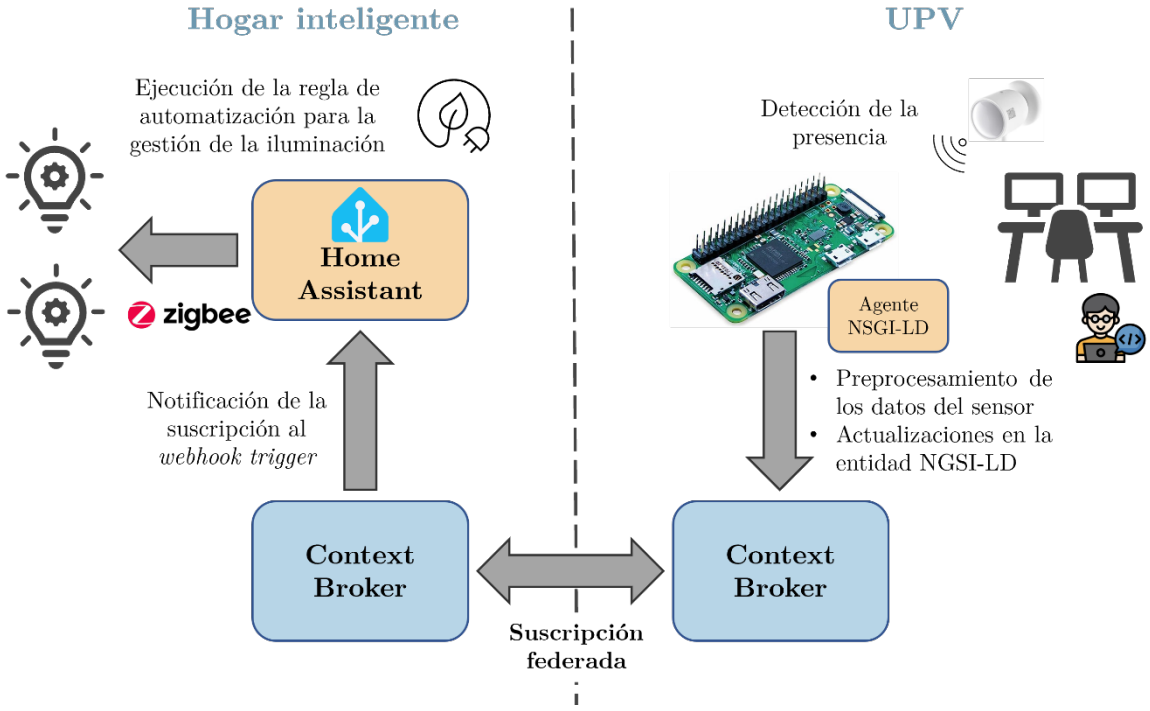


Figura 6.3: Escenario de validación del caso de uso del laboratorio

En resumen, este escenario refleja como un sensor IoT puede desencadenar acciones en un dominio externo sin necesidad de interactuar directamente con la infraestructura, o un servicio desplegado en ella, perteneciente al dominio en el que se realiza la acción. Además, tampoco se requiere un acceso físico o virtual a dicho dominio. Esta funcionalidad es posible gracias a la ubicuidad en el acceso a la información de contexto y a la federación entre dominios del continuo, como se ha detallado en el capítulo 5 de implementación de la tecnología.

6.3. Demostrador del proyecto aerOS

El segundo escenario de validación corresponde con una simulación completa de un despliegue prototípico y paradigmático del ámbito del IoT. En este escenario se combina el uso de infraestructura profesional, proporcionada por un proveedor polaco de servicios de computación en la nube (CloudFerro) [311], con infraestructura propia de un laboratorio de un grupo de investigación (el grupo de investigación público griego NCSRD), que a su vez mezcla equipamiento profesional

(servidores con capacidades de computación competentes) junto con elementos específicamente orientados a la experimentación y validación de tecnologías desarrolladas en el ámbito de la investigación, entre ellos una red 5G privada y local a la que se conectan dispositivos IoT.

El continuo de computación que forma la infraestructura utilizada para la implementación de este escenario validación se utilizó como base para la defensa del proyecto aerOS en Bruselas delante de unos evaluadores de la Comisión Europea, que finalizó con un resultado favorable, y causando una gran impresión a estos evaluadores. En esta evaluación se integraron los bloques esenciales del Meta-OS desarrollados en esta tesis doctoral dentro de todo el *stack* del proyecto aerOS para lograr el funcionamiento del flujo diseñado para dicha demostración, en la que se pretendía simular un caso de uso que fuera totalmente factible en un escenario real. Bajo el criterio del doctorando, este caso de uso debería ser clasificado con un nivel 5 de TRL, ya que se produjo una validación de la tecnología desarrollada en un entorno relevante y en condiciones que simulan su aplicación real, pero sin dejar de estar controlado.

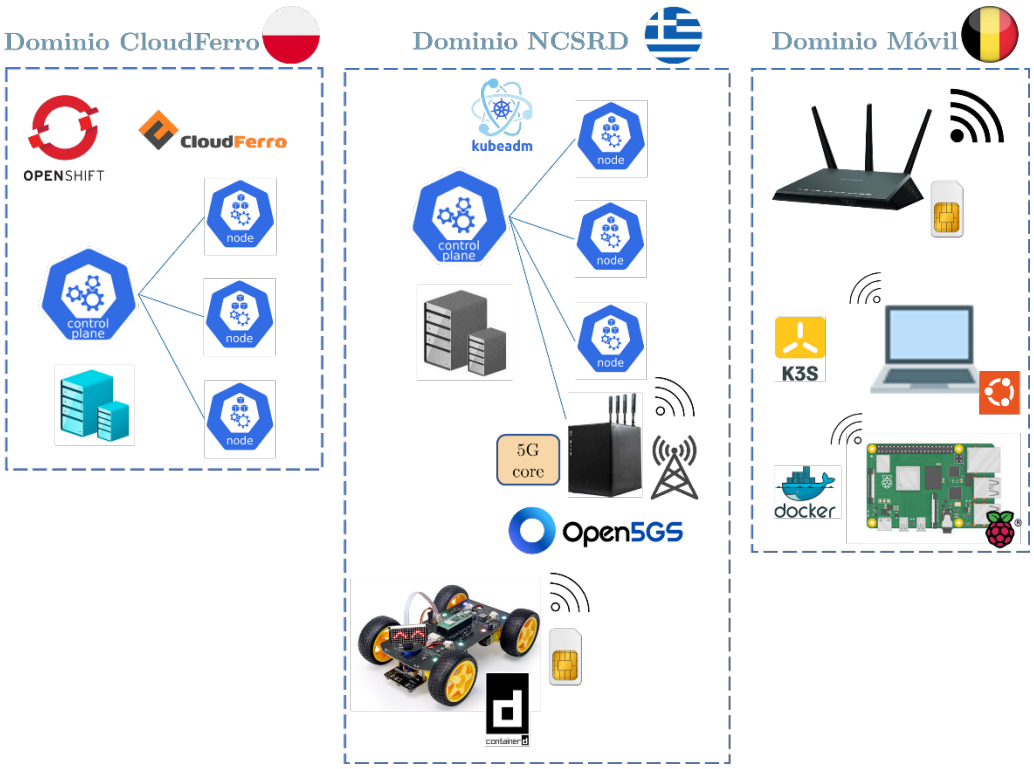


Figura 6.4: Continuo del demostrador del proyecto aerOS

La división inicial de dominios corresponde con la descripción inicial realizada en el primer párrafo. El primer dominio contiene un clúster de K8s desplegado utilizando el panel de usuario de CloudFerro siguiendo un modelo de KaaS (*Kubernetes as a Service*), que utiliza la tecnología OpenShift. Ha sido seleccionado como *entrypoint* debido a la fiabilidad que proporciona esta infraestructura al tratarse de un dominio *cloud* profesional. El segundo dominio incluye un clúster de

K8s *on-premises* junto con un dispositivo IoT (que cuenta con una gran variedad de sensores y actuadores) móvil simulando un vehículo, conectado a una red privada 5G. La topología interna de este dominio podría considerarse como *edge* o incluso *far-edge* debido a la inclusión de elementos IoT. Finalmente, el último dominio se trata de un dominio móvil que se desplegó en tiempo real *in-situ* en la misma sala donde se realizó la defensa ante la EC para demostrar la flexibilidad del Meta-OS en cuanto a la adición de nuevos dominios al continuo. Está compuesto por un ordenador portátil y una Raspberry Pi, que fueron conectadas a un router 4G para dotarlas de conexión a la red, demostrando la facilidad de la instalación del Meta-OS incluso en equipos con capacidades más limitadas de computación que son tan frecuentes. El criterio seguido para realizar esta división de dominios, aparte de reflejar el tipo de infraestructura utilizada, responde a una clara división geográfica.

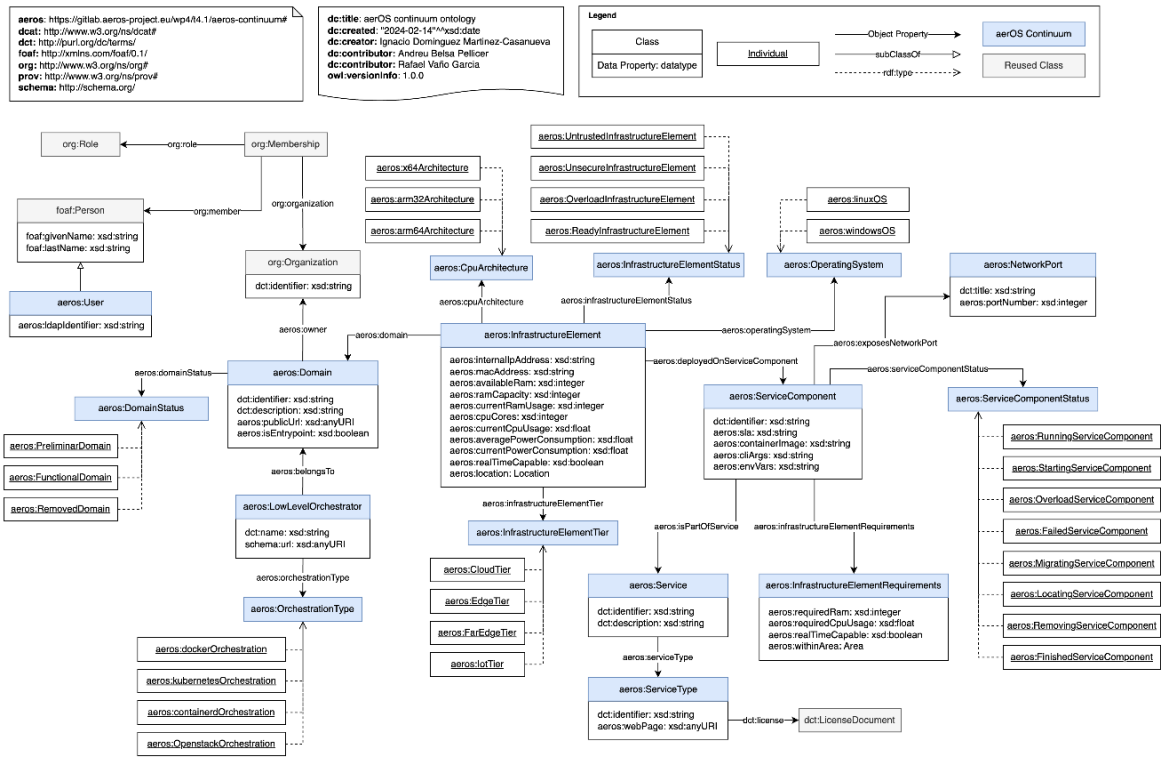


Figura 6.5: Ontología del continuo de aerOS [313]

Antes de proceder a la descripción del flujo del demostrador, es necesario destacar dos aspectos transversales a este caso de uso. En primer lugar, en el proyecto aerOS se decidió tomar como punto de partida el modelo de datos creado en esta tesis (descrito en la sección 5.2.1) para realizar una adaptación a sus propias necesidades. Esta adaptación consistió en la elección de las entidades básicas (*Domain, ComputingNode, ServiceComponent*...) y la posterior adaptación de su nomenclatura (por ejemplo, *ComputingNode* se convierte en *InfrastructureElement*). Además, dado que aerOS cuenta con paquetes de trabajo relacionados con la semántica de datos e implementa su propio *Data Fabric*, se construyó una ontología semántica a partir de esta adaptación del modelo de datos

aplicando la metodología LOT (Linked Open Terms) [312], cuyo resultado se puede comprobar en la Figura 6.5 [313]. Esta ontología constituye una demostración práctica de la flexibilidad de implementación del modelo de datos desarrollado en esta tesis, como se vislumbró en el apartado 5.2.1. Por otra parte, también es importante destacar que los componentes desarrollados durante el proyecto aerOS han sido empaquetados para K8s utilizando la herramienta Helm chart generator realizada por el doctorando (descrito en la sección 5.6.1), permitiendo agilizar el proceso de empaquetado y su integración en la metodología DevPrivSecOps desarrollada en el proyecto [265].

El flujo del demostrador realizado en Bruselas empieza con un continuo compuesto solamente por los dominios CloudFerro y NCSRD, girando en torno al dispositivo IoT desplegado en el dominio de NCSRD. Este dispositivo se trata de un pequeño vehículo construido en su propio laboratorio, que cuenta con una serie de sensores para medir ciertos parámetros relacionados con el coche (velocidad, detección de obstáculos, potencia de la señal recibida, ubicación, dirección...). Está conectado a una red 5G de experimental privada, cuyo 5G *core* basado en el *software* Open5GS [314] está desplegado en un nodo del clúster, y que utiliza el equipamiento AMARI Callbox Classic de la compañía Amarisoft [315].

Figura 6.6: Vehículo IoT utilizado en el demostrador de aerOS junto con la entidad NGSI-LD que lo representa

En primer lugar, se desplegó un servicio en el vehículo IoT que tiene dos funcionalidades claras: por una parte, recoger las mediciones realizadas por los sensores y actualizar su entidad correspondiente en el CB de su dominio, que sigue el formato del modelo de datos *Vehicle* de FIWARE Smart Data Models [316], como se puede comprobar en la segunda imagen de la Figura 6.6; por otra parte, llevar a cabo las órdenes de movimiento reflejadas en esta entidad (atributos *move, direction* y *heading*), suscribiéndose a esta entidad de NGSI-LD para recibir actualizaciones de la misma. Este despliegue se realizó utilizando el modo manual (recordemos que los modos de orquestación fueron descritos en el apartado 4.5.2), es decir, seleccionando el nodo correspondiente al vehículo, puesto que el servicio debe ejecutarse necesariamente en él. Esta orden de despliegue se materializó por

el LLO de tipo containerd previamente instalado en el mismo (desarrollado por el doctorando, ver sección 5.4.4.2), demostrando ser la mejor elección para un nodo de computación con unos recursos tan limitados que deben ser destinados casi por completo para su funcionalidad principal, siendo en este caso el control y la monitorización del vehículo.

En segundo lugar, se desplegó un servicio compuesto por dos componentes que tienen como fuente de entrada principal los datos de monitorización del vehículo, es decir, su información de contexto almacenada en el CB. El primero de ellos consiste en un pequeño módulo de IA que procesa los datos del vehículo para predecir su próxima posición óptima en un área predefinida, evitando posibles choques con obstáculos, y que finalmente envía la orden para que acuda a ella. El otro componente se trata de un panel de visualización de Grafana en el que se muestran los datos enviados por el vehículo en tiempo real. Este servicio fue desplegado dejando libertad al sistema de orquestación para que el HLO escogiera los nodos óptimos para su despliegue, resultando en dos nodos pertenecientes al dominio de CloudFerro, validando el sistema de orquestación federado entre diferentes dominios. En consecuencia, el funcionamiento de ambos componentes fue posible gracias el acceso ubicuo a esta información con independencia del dominio en el que estén desplegados.

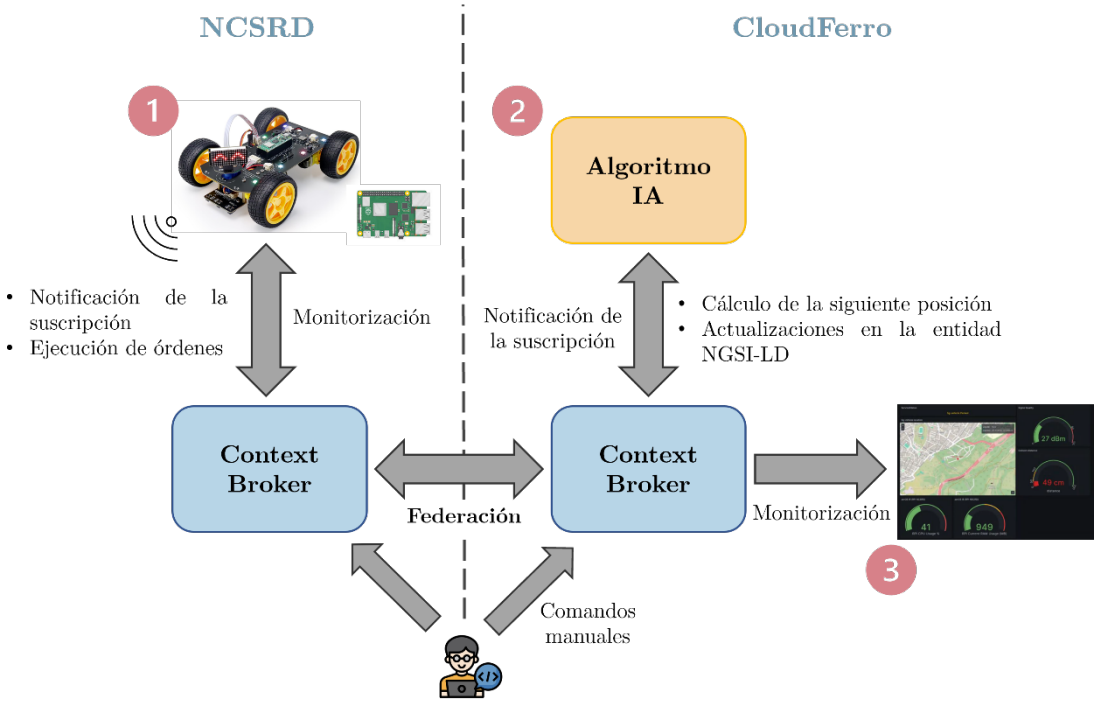


Figura 6.7: Primer flujo del demostrador de aerOS

Durante el demostrador se pudo comprobar mediante un vídeo en directo que el vehículo se movía gracias a los comandos enviados por el módulo de IA mediante los *context brokers* federados. Además, se enviaron comandos de manera manual a ambos CB para demostrar que el vehículo respondía a los mismos. Desafortunadamente, debido a que no es posible la grabación de imágenes en una

revisión de un proyecto de la EC por las estrictas regulaciones que se aplican, en la siguiente figura se muestra se muestra un ejemplo de la grabación realizada el día anterior para la preparación de esta demostración, en la que se realizó el mismo flujo [317]. En ella se puede observar el vehículo en movimiento, de manera que sus acciones se reflejan en un panel de Grafana en el que se muestran sus métricas en tiempo real, validando el flujo previamente descrito.

Figura 6.8: Validación del flujo del demostrador de aerOS

Una vez se demostró este primer flujo, en el que se validó el acceso a la información de contexto del vehículo con independencia del dominio, junto con el sistema de orquestación y monitorización del Meta-OS, se procedió a añadir un dominio que se desplegó en tiempo real *in-situ* en la misma sala en la que se realizó la defensa ante la EC para demostrar la flexibilidad del Meta-OS en cuanto a la adición de nuevos nodos de computación al continuo. Este dominio estaba compuesto por dispositivos simples y cotidianos como un ordenador portátil y una Raspberry Pi, a los que se dotó de conectividad mediante un router 4G instalado en la misma sala, pero suponiendo como contrapartida una ralentización de las comunicaciones necesarias para la federación entre dominios.

Después de añadir el nuevo dominio con éxito, como se pudo comprobar desde el portal de gestión desplegado en el dominio CloudFerro, se procedió a simular la sobrecarga del nodo en el que se encontraba desplegado el componente correspondiente al algoritmo de IA, con el propósito final de ordenar al HLO que desplegara el componente en otro dominio. Esta simulación se llevó a cabo utilizando el componente Self-Orchestrator de aerOS, que es capaz de detectar un umbral de uso de RAM en un nodo y ejecutar una función *serverless* para que ordene una reorquestación de los servicios desplegados en el nodo, permitiendo desacoplar la ejecución de esta función de un nodo que no es capaz de ejecutarla y transferirla a un nodo con mayores capacidades de computación. Esta interacción

demuestra la flexibilidad de los componentes básicos que habilitan el Meta-OS desarrollado para la integración de elementos adicionales que permiten ensanchar el ámbito de acción del Meta-OS para avanzar hacia una gobernación más eficiente del continuo de computación.

Finalmente, el HLO del dominio CloudFerro eligió desplegar el componente en el nuevo dominio móvil, demostrando que a pesar moverlo a un dominio completamente diferente sigue funcionando adecuadamente gracias a la federación entre dominios aportada por el Meta-OS.

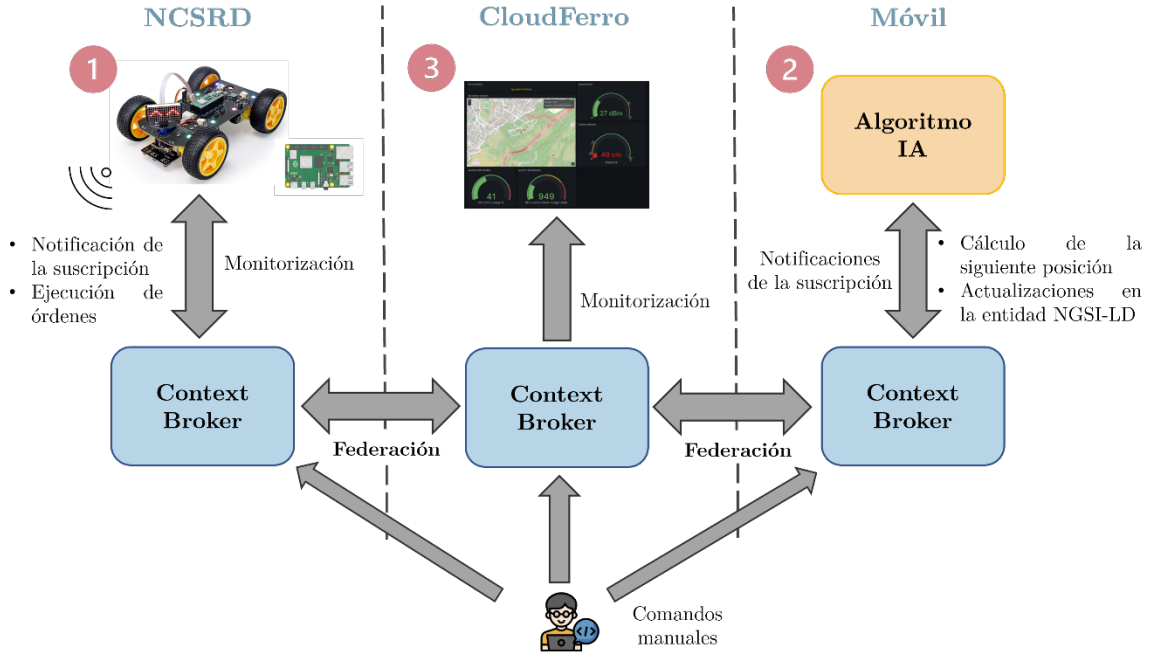


Figura 6.9: Segundo flujo del demostrador de aerOS

Asimismo, en todos los despliegues realizados por los diferentes LLOs del Meta-OS de este escenario de demostración se validó la integración de los LLOs con el sistema de fiabilidad del Meta-OS de aerOS, basado en la tecnología de DLT de código abierto IOTA. Esta funcionalidad fue introducida en la sección 5.7 como uno de los elementos adicionales al Meta-OS desarrollado en esta tesis, demostrando su capacidad de crecimiento y extensibilidad. De esta manera, toda acción realizada por cada LLO (creación, actualización o eliminación de un componente de un servicio) se introdujo como un nuevo mensaje en el *tangle* de IOTA, quedando la acción de orquestación registrada de manera inmutable, es decir, sin poder ser posteriormente alterada.

Figura 6.11: Mensaje registrado en el *tangle* de IOTA

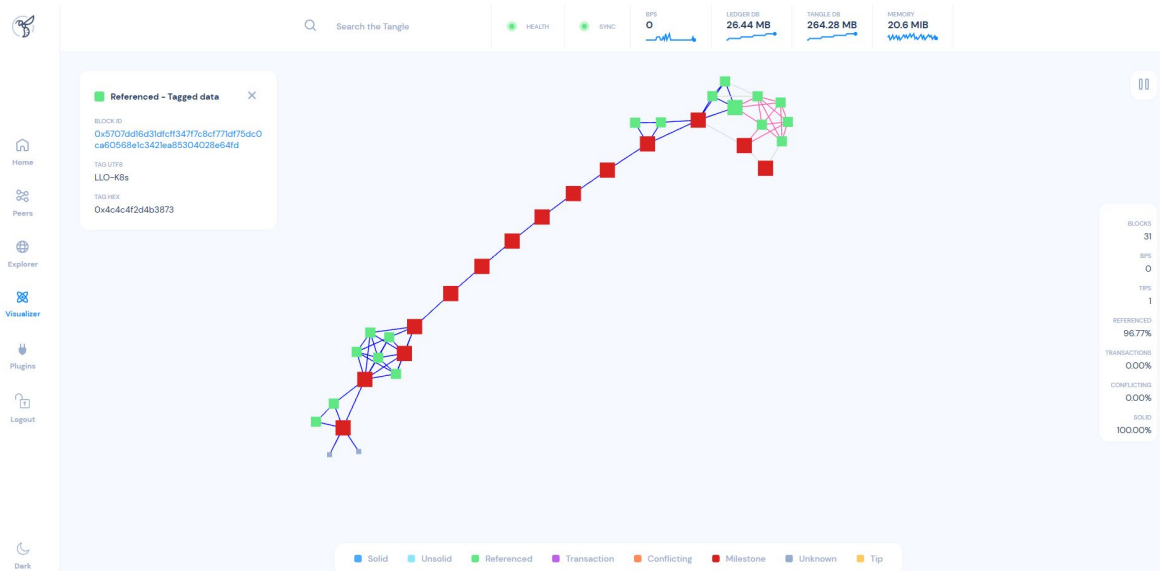


Figura 6.10: Panel de visualización de IOTA

Medidas del tiempo de respuesta en las comunicaciones propias de la federación entre dominios

De forma paralela a la realización de dos flujos de demostración descritos anteriormente, y aprovechando el continuo de computación desplegado para dicho propósito, el doctorando realizó unas pruebas para medir la latencia de las comunicaciones necesarias entre los *context brokers* de los dominios para el acceso ubicuo a la información de contexto del Meta-OS. El objetivo principal era la validación de la capacidad de escalabilidad del Meta-OS en cuanto a la adición de dominios y la posterior capacidad de federación entre ellos.

El valor de estas mediciones consistió en la realización de peticiones, realizadas a diferentes CBs del continuo, para obtener la información de contexto relacionada

con los nodos de computación del continuo, es decir, la obtención de las entidades NGSI-LD del tipo *ComputingNode*. No obstante, es necesario destacar que estas medidas son susceptibles a ser influenciadas por diferentes factores que no están estrictamente relacionados con el Meta-OS, como, por ejemplo, la latencia introducida por la propia red, la carga actual de la red, o el tiempo de procesamiento de los mensajes en los nodos.

Se diseñaron tres escenarios diferentes para obtener estos valores de medición:

1. La creación de varias instancias de Orion-LD en la misma VM, aprovechando las herramientas para la realización de tests funcionales aportadas por el mismo Orion-LD [318]. Este escenario, independiente del Meta-OS, pretende proporcionar el máximo teórico de instancias de Orion-LD que se pueden federar, ya que, al ser una prueba de ámbito local en la misma máquina, se consiguen evitar la mayoría de los factores externos, pero a su vez presenta una fuerte dependencia del *hardware* de la máquina en la que se realizan las pruebas. No se pretende validar el propio rendimiento de Orion-LD a nivel general, puesto que cuenta con sus propias pruebas de carga [319].
2. El escenario descrito en el caso de uso anterior (apartado 6.2). Este escenario contiene nodos conectados a la misma red interna de la UPV, pero utilizando diferentes tecnologías de acceso (Ethernet, Wi-Fi, redes virtualizadas...), y distribuidos entre diferentes localizaciones físicas de la universidad.
3. El escenario de este caso de uso (demostración del proyecto aerOS), puesto que cuenta con dominios dispersos geográficamente a través de toda Europa, junto con una gran variedad de nodos y tipos de red.

En todos estos escenarios se siguió un procedimiento similar: el punto de partida se define como un continuo compuesto por solamente dos dominios, es decir, dos instancias de Orion-LD (uno por dominio) con las debidas *Context Source Registrations* creadas automáticamente por el elemento Federador. Una vez realizadas las mediciones oportunas, un nuevo dominio se añade al continuo, lo que implica una nueva instancia de Orion-LD junto la actualización de las CSRs de los CBs existentes, y se repiten las mediciones.

Tabla 6.2: Resultados de las pruebas realizadas para medir los tiempos de respuesta en las comunicaciones propias de la federación entre dominios

Escenario	**Aspectos relevantes**	**Número de dominios**	**Tiempo de respuesta**
1	Tests funcionales de Orion-LD ejecutados en la misma máquina virtual (4 vCPUs y 8 GB RAM).	2	42 ms

1	Tests funcionales de Orion-LD ejecutados en la misma máquina virtual (4 vCPUs y 8 GB RAM).	10	48 ms
1	Tests funcionales de Orion-LD ejecutados en la misma máquina virtual (4 vCPUs y 8 GB RAM).	25	55 ms
2	Dominios conectados a la misma red interna de la UPV, cuyos nodos en los que se ejecuta Orion-LD están desplegados la misma infraestructura.	2	180 ms
2	Adición de un nuevo dominio conectado a red interna de la UPV, pero situado en una localización diferente y conectado a una subred independiente.	3	200 ms
2	Introducción de un nuevo dominio conectado a una red local de ámbito doméstico fuera de la red de la UPV.	4	410-450 ms
3	Dominios dispersos geográficamente por Europa.	2	500-600 ms
3	Introducción del dominio móvil conectado a un router 4G en el mismo sitio de las pruebas con un nivel de señal bajo.	3	En torno a 2 segundos, con algunos picos de 3 s

Con los resultados obtenidos, que se muestran en la tabla anterior, es posible extraer diversas conclusiones. En primer lugar, el escenario inicial demuestra que Orion-LD cuenta con la suficiente capacidad de soportar la federación de múltiples dominios (siempre y cuando la máquina cuente con unas capacidades de computación medias), o, en otras palabras, de numerosas instancias del CB, cumpliendo con lo especificado en el estándar NGSI-LD. En segundo lugar, esta funcionalidad tiene una fuerte dependencia de las condiciones de las redes de los dominios, validando la suposición inicial, ya que los sistemas federados implican un coste adicional en cuanto a la latencia en las comunicaciones, como se apuntó en la sección 4.5.1. Este hecho se puede comprobar en el caso del dominio móvil, en el que su conexión a Internet era realmente débil, aumentando el tiempo de respuesta en el acceso a la información de contexto de manera ubicua y, por lo tanto, reduciendo el rendimiento general del sistema. Por último, la federación de la información de contexto ha sido validada para unos escenarios heterogéneos, incluso en condiciones que no pueden ser directamente controladas por el administrador del Meta-OS, siendo el caso análogo para un dominio *edge* desplegado cerca del

ámbito de acción sin estar específicamente preparado para la instalación de infraestructura de computación, como, por ejemplo, en un campo agrícola.

6.4. Explotación de los elementos esenciales del Meta-OS para soportar la innovación en despliegues operativos reales

En esta subsección se va a abordar el tercer y último escenario de validación del Meta-OS desarrollado en esta tesis doctoral. Este escenario se subdivide a su vez en tres diferentes despliegues reales encuadrados en el proyecto aerOS que aplican a sectores verticales u operacionales completamente diferentes: puertos marítimos o terminales de contenedores, edificios inteligentes y líneas de producción industriales. Por lo tanto, encaja perfectamente en un nivel 6 de TRL, ya que la tecnología detrás del Meta-OS ha sido validada en condiciones de operación o funcionamiento real en una serie de pilotos aportados por agentes relevantes en los sectores descritos.

El propósito que se persigue al incluir estos despliegues consiste en demostrar que el objetivo específico de cada uno de ellos, que no está relacionado directamente con el Meta-OS desarrollado por el doctorando, ha podido cumplirse gracias al despliegue y uso de este Meta-OS para gestionar de manera eficiente el continuo que conforma la infraestructura utilizada en cada uno de ellos, dado que no hubiera sido posible sin la existencia de este elemento habilitador clave. Por ejemplo, en el caso de uso del edificio de oficinas inteligente (subapartado 6.4.1), el propósito final se basa en el despliegue de un algoritmo de IA inteligente para predecir el espacio de trabajo óptimo de un trabajador en base a ciertas condiciones medioambientales medidas por una serie de dispositivos IIoT instalados a lo largo de las oficinas. Sin embargo, para el despliegue, funcionamiento e interconexión de los componentes tecnológicos que lo componen, es necesario el uso de un elemento de abstracción y gestión por encima del continuo de computación: el Meta-OS desarrollado por el doctorando. Además, el doctorando ha participado activamente en la instalación y puesta en funcionamiento de los continuos de computación de todos los casos de uso descritos, lógicamente incluyendo la instalación y despliegue del propio Meta-OS expuesto en esta tesis.

Cada uno de los casos de uso cuenta con una estructura similar. En primer lugar, se realiza una descripción general del mismo, junto con material visual para evidenciar su realización. Posteriormente, se describen sus requisitos de actuación y los puntos claves para conseguir sus propósitos finales. En tercer lugar, se muestra un esquema de bloque en el que se ilustra su continuo de computación junto con los elementos claves del Meta-OS desarrollado. Finalmente, entrando en detalles más técnicos, se justifican los elementos claves del Meta-OS que soportan los

requisitos de cada piloto, validando la actuación y necesidad de inclusión de los mismos, junto con evidencias de la ejecución final de caso de uso en forma de materiales visuales. Desafortunadamente no se cuenta con resultados ni datos de validación finales puesto que el proyecto aerOS, en el que se enmarcan estos casos de uso, aún no ha llegado a su fin.

Por último, antes de proceder con cada uno de los casos de uso, y aunque no se vaya a describir con el mismo nivel de atención que éstos, es interesante destacar un caso de uso adicional que ilustra perfectamente el alto nivel de flexibilidad y de adaptabilidad con el que se ha dotado al diseño de los LLOs (apartado 5.4.4) del sistema desarrollado por el doctorando. Este caso de uso, dirigido por CloudFerro (mismo proveedor *cloud* que en la sección anterior), consiste en el despliegue de dos contenedores prefabricados que contienen un pequeño centro de datos en su interior, que es alimentado por completo por energías renovables: su única fuente de energía son una serie de placas solares instaladas en su techo.

Figura 6.12: Centro de datos dentro del contenedor prefabricado de CloudFerro

A nivel técnico, estos pequeños centros de datos contienen clústeres de K8s y cuentan con una funcionalidad que auto escala (aumenta o reduce el número de nodos de un clúster) estos clústeres de manera automática en base a la demanda de las capacidades de computación en tiempo real. Esta funcionalidad agrupa los nodos previamente en *nodepools*, que actúan como la unidad mínima de despliegue del sistema de orquestación del Meta-OS desplegado en este caso de uso, reemplazando al nodo de computación (o nodo de un clúster en el caso de K8s). Por lo tanto, cuando el HLO selecciona un nodo de computación para realizar el despliegue de un componente de un servicio, el LLO toma el *nodepool* al que está asociado (nuevo campo del modelo de datos) para que se despliegue en cualquier nodo perteneciente a este, ya que el nodo en concreto podría desaparecer al ser el *nodepool* reescalado).

De esta manera se consigue validar la adaptación del LLO de K8s, ya sea como un nuevo tipo de LLO o para como reemplazo de este, a los requisitos específicos de un caso de uso real.

6.4.1. Edificios de oficina inteligentes

Este caso de uso, propuesto por la compañía líder de telecomunicaciones griega OTE, pone su foco en una serie de edificios de oficinas inteligentes de la compañía situados en Atenas, que también han de ser energéticamente eficientes y sostenibles [320]. Busca integrar tecnologías IoT en entornos laborales con el objetivo de optimizar la eficiencia energética y garantizar la salud y seguridad de sus ocupantes mediante la toma autónoma y descentralizada de decisiones. El piloto implementa una infraestructura tecnológica que incluye sensores IoT, *gateways* multifuncionales, elementos de computación *edge* y una infraestructura *cloud on-premise* para gestión de dispositivos, almacenamiento y procesamiento de datos, abarcando el espectro completo del continuo *Cloud-Edge-IoT*. Este enfoque busca responder en tiempo real a cambios operativos y ambientales, ajustando automáticamente las condiciones en los espacios de trabajo y ofreciendo interfaces de usuario para monitorizar el consumo energético y parámetros clave de seguridad y salud.

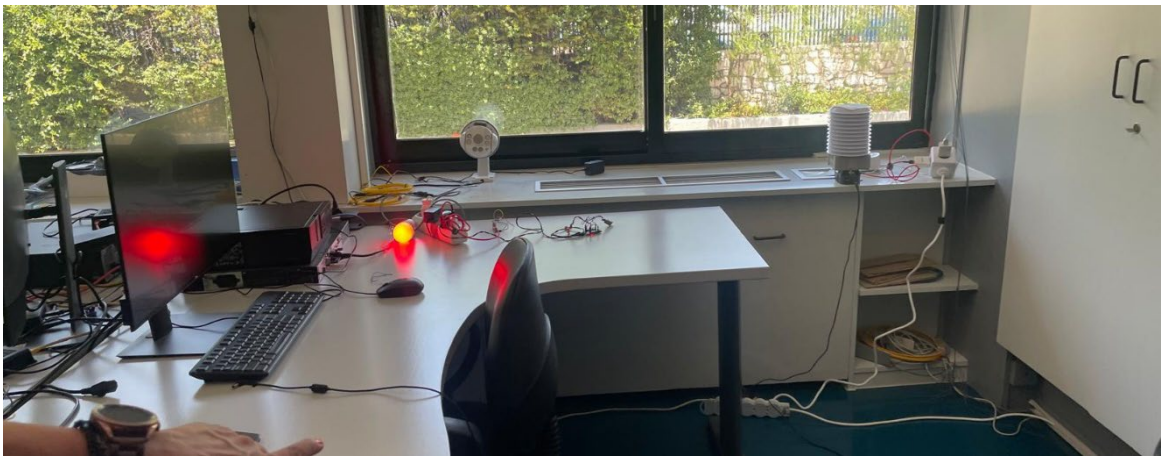


Figura 6.13: Puesto de trabajo de la oficina inteligente de OTE

Actualmente, los edificios de oficinas de las empresas se enfrentan a varios desafíos clave. En primer lugar, necesitan mejorar su eficiencia en el consumo energético, puesto que el precio de la energía ha aumentado considerablemente en Europa debido a la coyuntura sociopolítica del momento. Por otra parte, han de adaptarse a los nuevos modelos laborales híbridos impulsados después de la pandemia del COVID-19, en los que los empleados combinan el trabajo remoto y presencial. Esta rotación implica gestionar de forma flexible la asignación diaria de espacios laborales y asegurar condiciones seguras y saludables para los empleados en tiempo real. En este contexto, la integración efectiva de múltiples soluciones IoT heterogéneas representa una barrera técnica significativa, agravada por la necesidad

de gestionar grandes volúmenes de datos de manera local para que su proceso no sea centralizando, una de las grandes ventajas que aporta el paradigma del *edge computing* como se ha visto en esta tesis doctoral. Además, el procesamiento local de estos datos aumenta la seguridad de los mismos, un hecho en el que se necesita hacer hincapié debido al manejo de datos relacionados con la salud de los trabajadores.

Las fuentes de datos de este piloto incluyen sensores para la monitorización de valores ambientales (temperatura, luminosidad, calidad del aire, gases contaminantes, movimiento...), sus propios metadatos (batería, calidad de la red, disponibilidad...) y los datos anonimizados de los empleados con relación a la empresa (responsabilidades, ámbitos de actuación, lugar que ocupan en el organigrama...). Parte de los valores históricos de los datos proporcionados por los sensores son almacenados localmente en bases de datos de series temporales (InfluxDB [321]), también accesibles mediante APIs. Adicionalmente, se cuenta con interfaces de usuario orientadas a los gestores del edificio que permiten monitorizar el consumo energético y parámetros clave de seguridad y salud. Al mismo tiempo, este piloto incluye actuadores relacionados con el control de la ventilación, temperatura, sistemas de aire acondicionado e iluminación para asegurar las condiciones óptimas de trabajo.

Figura 6.14: Sensores de la oficina inteligente de OTE

Para la correcta ejecución de este caso de uso se deben cumplir una serie de requisitos. En primer lugar, la federación entre dominios del continuo facilita la integración y coordinación fluida de los recursos tecnológicos distribuidos en las diferentes oficinas de la compañía, ya que se acordó que cada edificio constituyera un dominio del continuo. Dicha federación permite gestionar elementos de computación y dispositivos IoT heterogéneos provenientes de diferentes fabricantes mediante una capa común de interoperabilidad, superando así la barrera relacionada con la diversidad de soluciones propietarias y aisladas que complican la comunicación e integración entre sistemas. Además, permite la adición de nuevos edificios al sistema de manera transparente para los administradores de sistemas, gracias al elemento Federador. Esta flexibilidad también se plasma cuando ocurren

modificaciones en la distribución física del entorno productivo, reduciendo el tiempo requerido para reasignar o reconfigurar equipos.

El acceso ubicuo a la información de contexto proporcionado por el Meta-OS es esencial para responder eficazmente al requisito de optimización dinámica del diseño y asignación de los recursos productivos. Esta funcionalidad permite que los datos generados por sensores IoT, sistemas de actuación, así como información clave del continuo (por ejemplo, monitorización en tiempo real del estado operativo de nodos de computación y recursos energéticos) estén accesibles de manera transparente desde cualquier edificio de oficinas sin depender de la transmisión centralizada de grandes volúmenes de datos. De esta manera, se asegura el requerimiento del piloto para ofrecer respuestas rápidas, descentralizadas y automatizadas, logrando así una reducción en tiempos de respuesta, menores costos operativos y una menor dependencia de intervención manual.

Finalmente, el diseño del sistema de orquestación federado del Meta-OS permite soportar el requisito escalabilidad relacionado con el despliegue de múltiples aplicaciones IoT en las oficinas, especialmente cuando el número de edificios y oficinas aumente, con la asociada inclusión de nuevos dominios. Asimismo, habilita la configuración de flujos de datos IoT para que sean procesados de manera local por los servicios requeridos, o que respondan a órdenes de actuación emitidas por servicios desplegados en cualquier nodo del continuo. También es necesario destacar que el LCM proporcionado por el sistema de orquestación del Meta-OS permite controlar el estado de los servicios desplegados en varios niveles, así como detectar posibles fallos para emitir órdenes de reorquestación de servicios para cumplir con el requisito del piloto relacionado con la recuperación automática de servicios en el momento en que haya fallos en el sistema, tanto a nivel de red como de nodo.

Como se puede observar en la Figura 6.15, se decidió crear un dominio *entrypoint* que alojara las aplicaciones operativas específicas (algoritmos de AI, aplicación web de los empleados, etc.) y un dominio para el propio edificio de oficinas que contiene los dispositivos IoT, cuyos datos son gestionados mediante el uso de Home Assistant (al igual que en el escenario del laboratorio SATRD), junto con equipamiento computacional de mayor potencia para alojar los componentes claves del Meta-OS. Aunque el piloto cuente por el momento con un único edificio de oficinas, la intención de OTE es escalar estos dominios con la adición de nuevos edificios, dotando de flexibilidad a los administradores para distribuir las aplicaciones generales desplegadas en el primer dominio. Además, al tratarse de una compañía de telecomunicaciones, cada edificio y dominio cuenta con su propia IP pública, facilitando las comunicaciones para la federación de dominios.

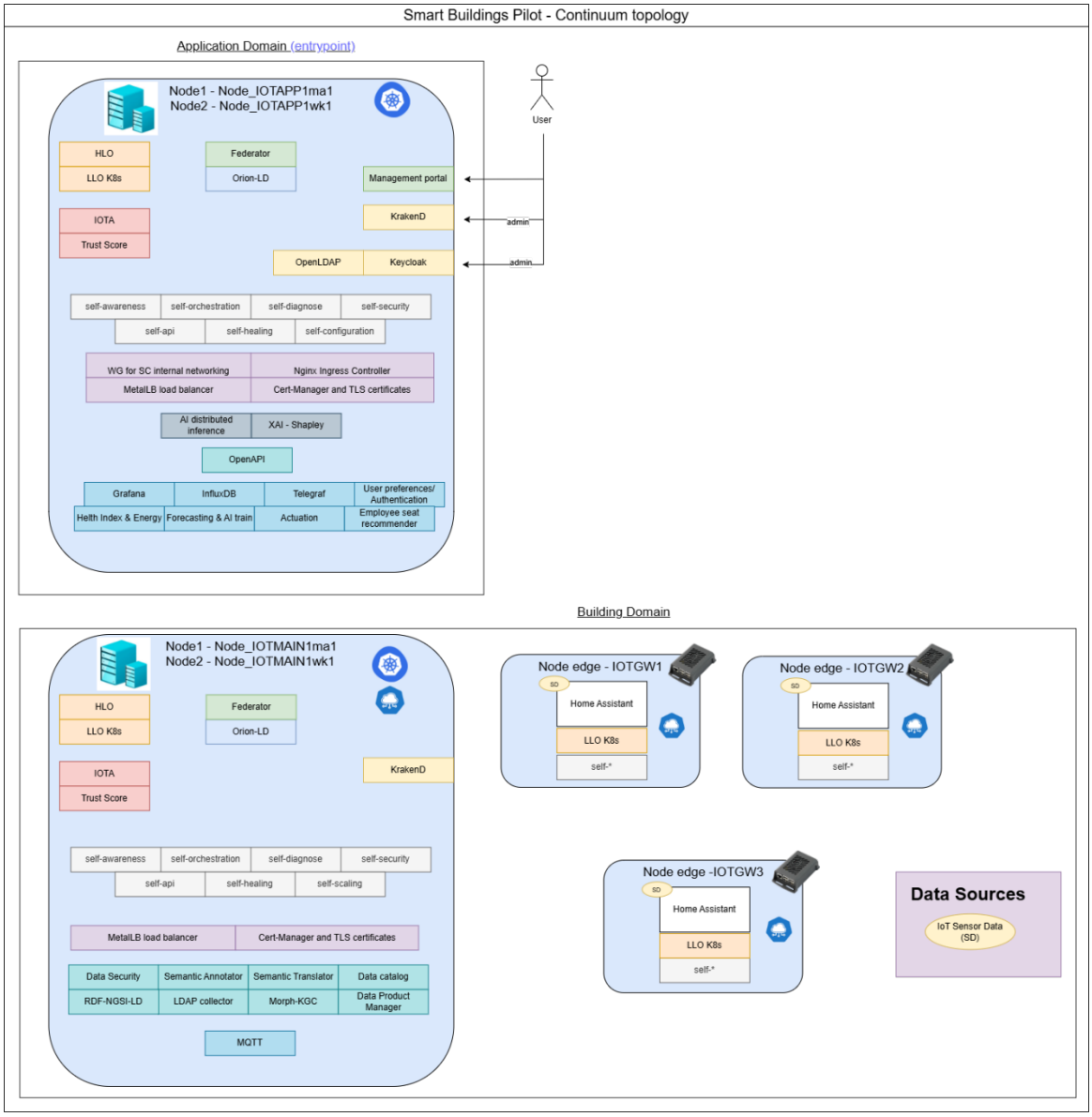


Figura 6.15: Continuo del caso de uso de edificios de oficinas inteligentes

A nivel técnico, el resultado principal de este caso de uso se trata del desarrollo de un algoritmo de IA que, tomando como datos de entrada las condiciones ambientales de las oficinas, sus datos de ocupación y de los empleados, muestre a un empleado en concreto el sitio óptimo que debe ocupar para optimizar la eficiencia energética del edificio y las condiciones saludables del entorno. Además, este algoritmo tiene que proporcionar una estimación del consumo de energía y de las condiciones ambientales de cada oficina [322]. Estos resultados, además de la monitorización de las oficinas en tiempo real, se visualizan en una aplicación web propia, desplegada en el dominio de aplicaciones. Aunque el desarrollo del algoritmo de IA y la interfaz mostrada a continuación no han sido realizadas por el doctorando en el ámbito de esta tesis, su ejecución es posible debido al Meta-OS implementado por el doctorando.

Figura 6.16: Visualización del estado de las oficinas de un edificio de OTE

Un requisito de diseño técnico de importante era el uso de K8s como sistema de orquestación de contenedores, debido a la política tecnológica de la compañía, y el desarrollo de ciertos componentes que solo eran compatibles con K8s. Por este motivo, el doctorando eligió KubeEdge debido a su propia arquitectura adaptada al *edge*, mediante la cual se puede instalar un pequeño clúster de K8s en cada edifico para que actúe como nodo *cloud*, y desplegar en la IoT *gateway* instalada en cada oficina su parte *edge* para poder unirse al clúster. Este diseño es posible gracias al LLO de K8s, que es compatible con cualquier distribución de K8s. Además, es necesario destacar que esta solución demuestra que una solución con su propia arquitectura *Cloud-Edge*, como KubeEdge, puede integrarse sin ningún problema ni configuración adicional en el Meta-OS diseñado.

Mientras que en el dominio del edificio se recolectan los datos de los sensores, se integran en Home Assistant y se inyectan en el CB de su dominio para que estén disponibles por los otros dominios gracias a la federación del Meta-OS, en el dominio de general de aplicaciones se accede a los mismos de manera federada para alimentar el algoritmo de predicción y toma de decisiones. De esta manera los datos permanecen en un ámbito local, restringiendo el acceso a los mismos y aumentando su seguridad y privacidad, ya que solamente se almacenan algunos datos anonimizados en una base de datos histórica en forma de series temporales (InfluxDB) para el posterior entrenamiento y mejora de dicho algoritmo. Además, estos datos de los sensores son procesados en paralelo en servicios desplegados en el mismo dominio del edificio para realizar las actuaciones necesarias para mantener unas condiciones ambientales aceptables en la oficina, aprovechando el sistema de suscripciones de Orion-LD.

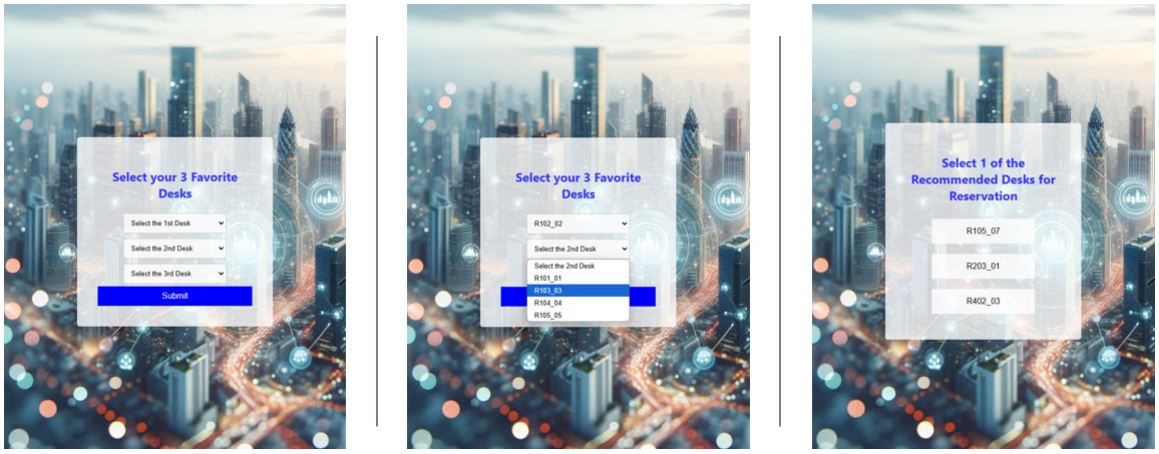


Figura 6.17: Visualización del sistema de recomendación de puestos de trabajo

6.4.2. Terminal de contenedores portuaria

Este escenario de validación se desarrolla en la terminal de contenedores (EGCTL) del puerto de Limasol (Chipre), operada por la empresa EUROGATE. Este puerto es el más grande de la isla, dado que absorbe el 90% del tráfico total de contenedores (aproximadamente 300.000 TEUs —*twenty-foot equivalent unit*— anuales), demandando una operación continua durante las 24 horas de todos los días del año. El equipamiento de esta terminal incluye más de 35 grúas de pórtico sobre neumáticos (RTG), 5 grúas de pórtico para contenedores (STS , siendo dos del tipo Super Post Panamax que pueden descargar 2 contenedores simultáneamente), 5 vehículos para el manejo de contenedores vacíos y 10 tractores de terminal o de carga. Además, las operaciones realizadas en la terminal, muchas de las cuales son críticas, son gestionadas en tiempo real por el sistema operativo de terminal (TOS), que ha sido desarrollado y es mantenido por EUROGATE, e integrado con el PCS del puerto. En este contexto, EUROGATE está involucrado en el proceso de digitalización y automatización de estos procesos operacionales, en los que ha tenido también una participación clave la empresa española Prodevelop, encargada de la instalación de la plataforma IoT de la terminal.

Actualmente, existen diversas problemáticas y obstáculos operativos que limitan significativamente la eficiencia en la gestión de contenedores en el puerto de Limasol. Uno de los desafíos más críticos es la dependencia de procesos manuales para la inspección visual de daños en contenedores y la identificación de sellos rotos o incorrectos. Estos procedimientos manuales, además de ser lentos, tediosos y propensos a errores humanos, incrementan considerablemente los tiempos operativos, generando demoras que afectan la productividad global del puerto. Asimismo, estas inspecciones manuales conllevan riesgos de seguridad significativos para el personal que opera cerca de equipos de gran tonelaje y peligrosos (por

ejemplo, contenedores apilados), aumentando la posibilidad de que ocurran accidentes laborales [323].

Otro problema relevante es la carencia de mecanismos automatizados para la detección de forma temprana de daños o fallos en el equipamiento operativo esencial de la terminal, como las grúas de contenedores o los vehículos de transporte. La identificación temprana de estas anomalías permitiría adaptar los procesos de mantenimiento correspondientes para evitar los paros frecuentes en la actividad como consecuencia de esta detección tardía, y en consecuencia las pérdidas económicas generadas por los retrasos en la carga y descarga de las mercancías. Además, la falta de una integración efectiva de datos provenientes de diferentes fuentes impide la realización de cualquier análisis predictivo de manera agregada o general.

Figura 6.18: Terminal de contenedores del puerto de Limasol (EGCTL)

El objetivo principal de este piloto es resolver estos desafíos mediante el desarrollo e implementación de soluciones basadas en visión por computadora o artificial (CV, como el reconocimiento automático de objetos en imágenes), Inteligencia Artificial (IA) y tecnologías avanzadas IoT integradas en el continuo de computación CEI. Mediante el uso de estas tecnologías está previsto que se reduzcan drásticamente los tiempos de inspección de contenedores, se minimicen errores humanos, y se optimicen las operaciones de mantenimiento la maquinaria.

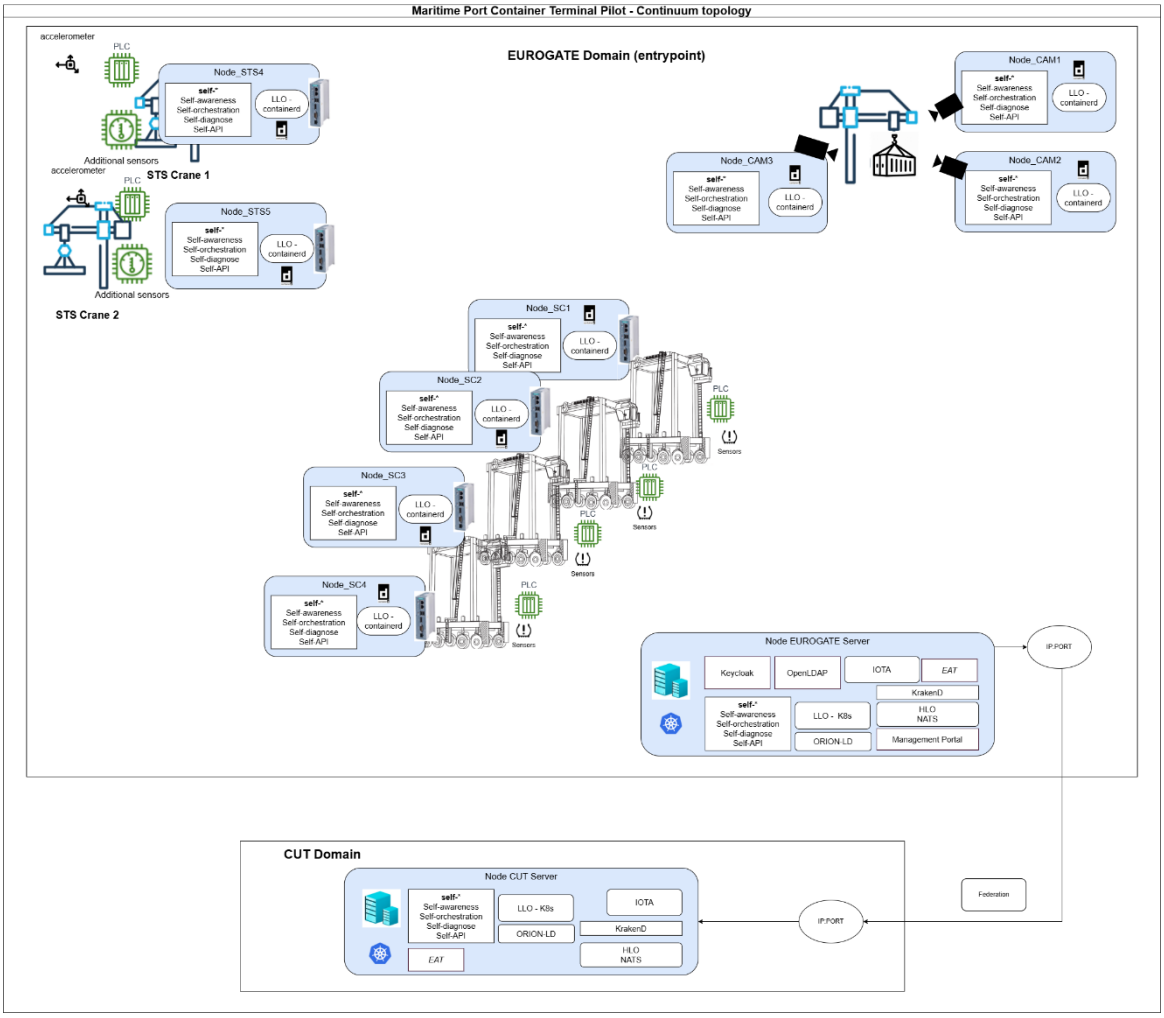


Figura 6.19: Continuo del caso de uso de la terminal de contenedores portuaria

Debido a los requerimientos técnicos y funcionales de la terminal de contenedores, los dominios de este piloto habían de dividirse con criterios eminentemente geográficos y operacionales puesto que los puertos y sus terminales forman entidades cerradas que son muy celosas de su privacidad. De esta manera, el dominio *entrypoint* está formado por los nodos de computación situados en la terminal, que comprenden pequeños elementos de computación instalados en las mismas grúas y cerca de las cámaras de grabación de los contenedores para la detección , además de un servidor desplegado en el mismo puerto. Este dominio no cuenta con ninguna IP pública ni expone ningún servicio al exterior, ya que las terminales como norma general suelen restringir el acceso desde la red pública a sus servicios. Por ende, el portal de gestión del continuo solo es accesible por los operarios IT desde la red interna de la terminal.

El primer paso consistió en cumplir el requisito preliminar, que consistía en la integración en el Meta-OS de (i) los sensores IoT instalados en las grúas y (ii) una serie de cámaras instaladas para la grabación automática de los contenedores sobre los que operan las grúas. La integración de los sensores de las grúas se consiguió

mediante el despliegue del mismo servicio, encargado de recoger los datos de los sensores e introducirlos en el CB del dominio, en cada uno de los dispositivos de computación adyacente a cada grúa gracias a la funcionalidad del despliegue por flotas proporcionada por el sistema de orquestación del Meta-OS (introducido en la sección 5.4.3.1). De manera adicional, este servicio se encarga de alimentar a un módulo de IA frugal —es decir, eficiente en el uso de recursos— desarrollado por la universidad chipriota CUT [324] (Cyprus University of Technology), y también desplegado en cada grúa gracias a su citada frugalidad, el cual realiza una serie de predicciones sobre sus necesidades de mantenimiento. Asimismo, la integración en el Meta-OS de estos elementos permitió monitorizar el estado de los elementos de computación adyacentes a las grúas y cámaras para dotar de una mayor capa de observabilidad a estos elementos (detallado en el apartado 5.2), y así detectar de manera preventiva fallos en los mismos, en sus sensores o en sus conexiones.

Con el propósito de sustituir la inspección manual de los contenedores y la grabación manual de los mismos, se desplegó una aplicación en cada uno de los dispositivos *edge* adyacente a cada cámara para la transmisión de las imágenes grabadas por estas al pequeño servidor local del puerto. Estas imágenes sirven para alimentar a un algoritmo de IA para la detección de fallos en los contenedores [325], desplegado en el mismo servidor ya que cuenta con unos requerimientos computacionales más elevados que el de las grúas, imposibilitando su despliegue directo en cada uno de los dispositivos adyacentes a las cámaras.

Figura 6.20: Fallos en los contenedores detectados por el módulo de IA

Ambas integraciones validan la necesidad del piloto de distribuir la potencia de computación, ya que previamente a la integración del Meta-OS, los dispositivos situados junto a las grúas enviaban toda su información a un potente servidor centralizado desplegado en la nube pública, en los que se realizaban las operaciones necesarias. Las mejoras aportadas por la introducción del paradigma del continuo de computación junto con el Meta-OS para su gobernanza eficiente, logran reducir

la latencia de las comunicaciones, reducir los costes operacionales y la probable aparición de cuellos de botella en el procesamiento de datos, debido al hecho que los procesos computacionales ocurren cerca de la fuente de datos y del ámbito de acción. Asimismo, se abre a la puerta de la utilización de algoritmos de IA basados en *Federated Learning* (FL), puesto que cumplen con el requisito de garantizar la privacidad de esta información, manteniéndola en el dominio local en todo momento.

Entrando en detalles sobre el despliegue de los servicios, tanto de recolección de datos como de los propios algoritmos de IA, se optó por el uso del LLO de containerd para los dispositivos adyacentes a las grúas y a las cámaras. Esta elección se sustenta en que, en este entorno portuario, donde existen grandes estructuras metálicas como grúas de gran tamaño y numerosos contenedores apilados, las conexiones inalámbricas a la red son poco fiables y robustas, en parte como consecuencia del efecto conocido como jaula de Faraday. En este tipo de LLO se asegura la entrega de los mensajes que contienen las órdenes de despliegue gracias a NATS JetStream (ver 5.4.4.2), ya que son retenidos en el *broker* hasta que son entregados al controlador del LLO y este confirma su recepción, permitiendo superar las posibles interferencias o cortes en las comunicaciones. Además, gracias al LCM del sistema de orquestación del Meta-OS, un administrador puede percatarse de estos cortes en las comunicaciones (el estado del componente se quedaría en *Deployed*). Esto es uno de los motivos por los que descartó el uso de KubeEdge como en el caso de uso de los edificios inteligentes previo, sumado a que el LLO de containerd es extremadamente ligero y permite destinar la práctica totalidad de los recursos de computación (ya de por sí limitados) a la ejecución de sus servicios básicos (el módulo de recolección de datos y el módulo de IA frugal).

Por último, el dominio de CUT es externo al puerto, estando situado en la propia universidad, y tiene como propósito único el desarrollo y entrenamiento de los modelos de IA citados. Estos modelos han sido entrenados a partir de una serie de datos específicos para dicho propósito proporcionados por la terminal, dado que un requerimiento del piloto era que estos datos reales solo pudieran ser accesibles desde el dominio de la terminal. Como la terminal era celosa de compartir estos datos mediante la habilitación de un *endpoint* público, utilizando la plataforma de gestión del dominio EUROGATE (el *entrypoint*), se desplegó un servicio que se encargaba de recolectar los datos de monitorización de los sensores de las grúas del CB del dominio junto con otros datos clave del TOS, los anonimizaba, y los inyectaba en el CB de la terminal permitiendo su acceso de manera federada desde el CB del dominio de CUT gracias a la configuración realizada en el Federador. Para conseguir establecer la federación entre ambos dominios, puesto que, como se ha comentado previamente, la terminal no deseaba exponer ninguna IP ni servicio públicamente, el dominio CUT se configuró como dominio *relay* al contar con una dirección IP pública (proceso que fue descrito en el apartado 5.3.2), y por lo tanto accesible por el domino EUROGATE.

6.4.3. Línea de producción de drones

El último caso de uso tiene lugar principalmente en la línea de producción de drones del laboratorio para la investigación de la Industria 4.0 del Switzerland Innovation Park (SIPBB), situado en la ciudad de Biel (Suiza). Este escenario aborda desafíos ambientales críticos mediante la integración de tecnologías avanzadas para enfrentar dichos retos, que incluyen la monitorización en tiempo real de emisiones de CO_2 asociadas a la fabricación de drones, la implementación de un Pasaporte Digital de Producto (DPP) y la optimización dinámica de procesos productivos y logísticos. El uso de estas tecnologías busca avanzar hacia una producción más eficiente y sostenible, asentando las condiciones necesarias para la creación de un modelo replicable basado en la economía circular y la fabricación personalizada siguiendo criterios medioambientales, disminuyendo así el impacto ecológico y mejorando la sostenibilidad global del entorno industrial [326].

El citado DPP (*Digital Product Passport*) que se pretende implementar tiene como propósito monitorizar y predecir de forma precisa la huella de carbono individual de cada producto fabricado, considerando variables como los materiales empleados, las condiciones ambientales de la línea de producción, la energía consumida y las emisiones emitidas por las máquinas que intervienen en el proceso de producción, personalizaciones específicas y métodos de entrega seleccionados por el cliente [327]. De esta manera, el DPP posibilitaría a los clientes estar informados de antemano sobre la huella de carbono de los productos demandados, y realizar adaptaciones en ellos para reducirla en base a esta predicción, avanzando hacia una producción más sostenible.

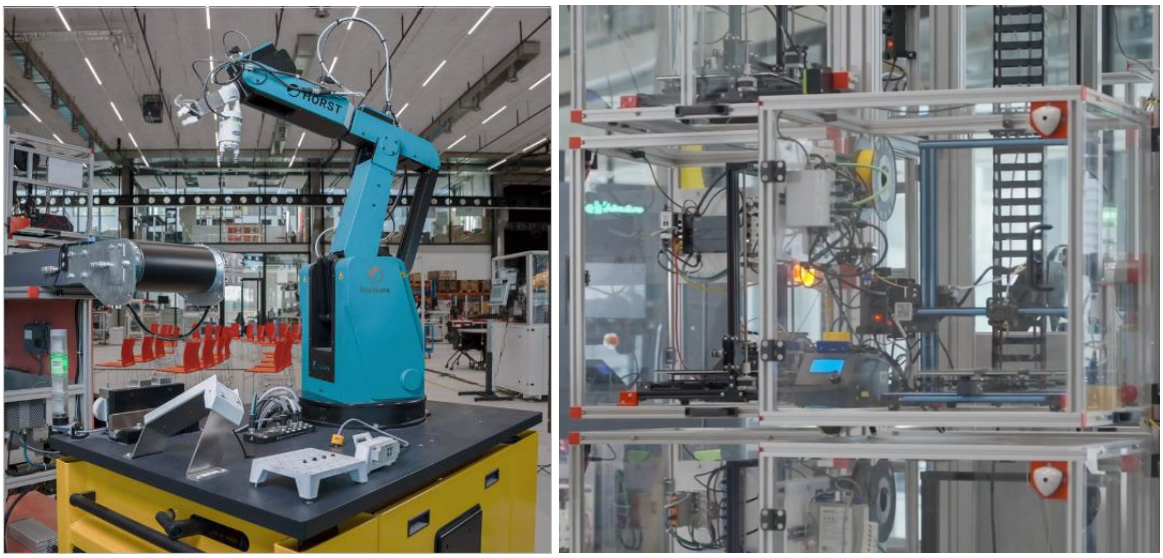


Figura 6.21: Línea de producción de drones del laboratorio de Industria 4.0 del SIPBB

La línea de producción de drones del laboratorio del SIPBB cuenta una gran heterogeneidad en cuanto a los dispositivos industriales que intervienen en ella, como estaciones manuales digitalizadas SETAGO, cintas transportadoras inteligentes, sistemas automatizados de soldadura de PCB, vehículos autónomos guiados (AGVs), y estaciones avanzadas de control de calidad equipadas con tecnologías IoT. La gestión unificada de esta heterogeneidad es uno de los principales desafíos de este caso de uso que se pretende solucionar mediante el uso de los componentes esenciales de un Meta-OS desarrollados en esta tesis, puesto que, aunque las máquinas sean dependientes entre ellas e intercambien los datos necesarios para el correcto funcionamiento de la línea de producción, estos datos no son procesados ni analizados de una manera unificada que permita tener un conocimiento global del proceso de producción de los drones. Además, esta integración permitiría integrar datos de proveedores externos.

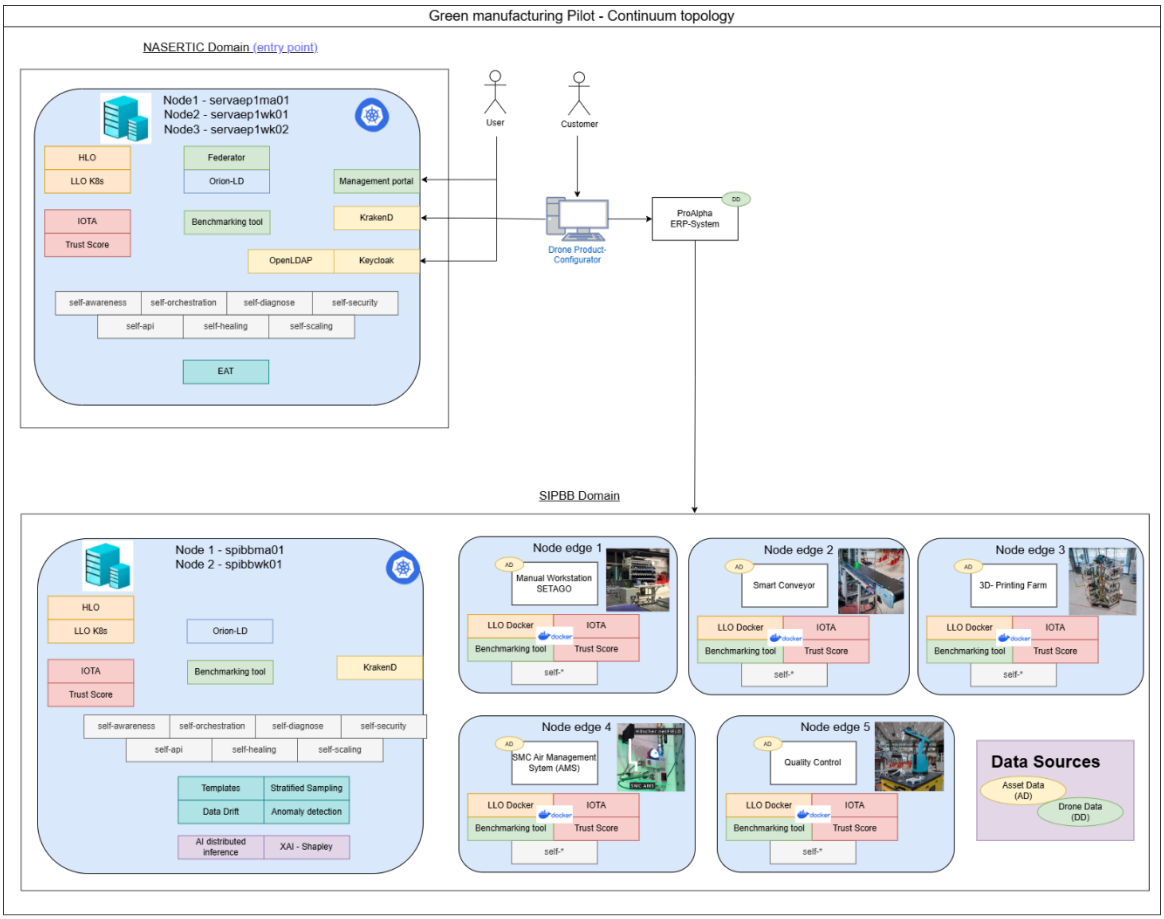


Figura 6.22: Continuo del caso de uso de la línea de producción de drones

Por otra parte, NASERTIC es un proveedor de servicios *cloud* público dependiente del gobierno de la Comunidad Foral de Navarra, que pretende ofrecer sus servicios de computación en la nube a empresas industriales para validarlos en entornos industriales antes de que puedan ser ofrecidos a empresas del sector industrial navarro, como parte de la estrategia de sostenibilidad y digitalización

implementada por el gobierno de Navarra. Esta infraestructura *cloud* se aprovechó para desplegar el dominio *entrypoint* debido a su propia condición *cloud*, en el que además se desplegaron servicios de ámbito global como el configurador de la producción de drones. De esta manera, al utilizar un dominio accesible públicamente a través de Internet, se logra evitar que los clientes accedan directamente a la infraestructura de la planta de producción, así como habilitar el control de dicha planta por parte de sus operarios desde un único punto externo a la misma.

La integración de los dispositivos y máquinas industriales en el continuo de computación ha de realizarse necesariamente mediante su conexión con elementos de computación que cumplan con los requisitos para ser considerados un nodo de computación del continuo (especificados en la sección 4.2), de manera que queden asignados a los mismos para ejercer un control sobre ellos. Los nodos de computación que controlan los dispositivos industriales tienen una naturaleza y condiciones muy específicas, ya que han sido creados especialmente para dicho propósito. Esto se traduce en que estos nodos tienen instalado un sistema operativo (frecuentemente basado en Linux) adaptado y con unas opciones de configuración muy limitadas, además de contar con una serie de *drivers* o controladores específicos, instalados por defecto para el control de dichas máquinas [45]. Asimismo, el sistema de gestión de contenedores (incluso algunos elementos de computación no cuentan con uno, en cuyo caso no podrían ser considerados nodos del continuo) que integran es la versión de Docker disponible en el momento de su producción, que en algunos casos podría resultar obsoleta.

La naturaleza tan cerrada de estos nodos hace que frecuentemente la actualización de sus características o funcionalidades más básicas, relacionadas con el OS o el mismo sistema de gestión de contenedores, no sea posible. Por lo tanto, el LLO de Docker desarrollado como parte del Meta-OS (ver apartado 5.4.4.2) es absolutamente necesario para habilitar la integración de estos nodos de computación (junto sus máquinas adyacentes) en el continuo, puesto que permite el despliegue de servicios que permitan gestionar eficientemente los datos aportados por los dispositivos industriales y ejercer un control sobre ellos, todo ello desde el portal de gestión unificada del Meta-OS. Mediante estos servicios desplegados por el LLO de Docker, los datos se pueden introducir en el CB del dominio de la planta industrial, estando disponibles por los servicios requeridos desplegados en el continuo (monitorización, análisis, etc.) para avanzar hacia un mayor conocimiento global del proceso de producción de los drones, cumpliendo con uno de los requisitos claves del caso de uso.

Entrando en detalles, estos datos de monitorización sirven para alimentar el algoritmo de IA clave para calcular la huella de carbono resultante de la fabricación de los drones y habilitar la creación del DPP de cada dron de manera individualizada [328]. Además, gracias a la federación entre dominios y sus CBs proporcionada para el Meta-OS, esta información podría ser compartida entre varias plantas de producción del mismo parque tecnológico con el propósito de

identificar patrones de comportamiento que aumenten el consumo de energía o la emisión de gases contaminantes, e incluso la retroalimentación de los algoritmos de IA de los DPPs correspondientes.

Figura 6.23: Monitorización de las condiciones ambientales sobre las que operan las máquinas de la planta de SIPBB

El intercambio de este tipo de información figura como uno de los requisitos del caso de uso, cuya ambición no se reduce a un intercambio de datos limitado al ámbito local de SIPBB. Es más, la intención de SIPBB es la inclusión de estos datos en espacios de datos colaborativos (conocidos como *data spaces*) a un nivel industrial europeo para avanzar en la reducción del consumo de energía y de la emisión de gases contaminantes. El uso de Orion-LD como CB del Meta-OS facilita sin lugar a duda esta integración [329], puesto que el componente Data Space Connector del ecosistema de FIWARE integra Orion-LD como elemento clave para el intercambio de datos en un *data space*, y es compatible con los estándares de las iniciativas para mejorar la soberanía e interoperabilidad de los datos Gaia-X, IDSA (International Data Spaces Association) y DOME (Distributed Open Marketplace for Europe) [330]. Además, FIWARE junto con Gaia-X, BDVA (Big Data Value Association), e IDSA, lanzaron en 2021 la Data Spaces Business Alliance (DSBA) para avanzar en esta materia y acelerar la transformación de la industria respecto al tratamiento e intercambio de datos [331].

Por último, la integración del dominio *cloud* con la infraestructura de NASERTIC permite habilitar la gestión unificada de varias líneas de producción desde un único punto, permitiendo a los operadores más técnicos controlar y monitorizar los elementos de las líneas de producción sin requerir su presencia física en las plantas, teniendo en cuenta que su localización está distribuida. Además, el despliegue del Meta-OS permite reducir el volumen de los datos transmitidos a este punto central, ya que solamente se envían datos a la plataforma de gestión bajo demanda o cuando es estrictamente necesario, debido a la propia arquitectura del acceso ubicuo a la información de contexto (detallada en el apartado 5.3.1). De no ser así, se transmitirían datos a la nube cada vez que se actualizara el estado de

una máquina, resultando en un uso ineficiente de la red. La propia naturaleza *cloud* de esta infraestructura habilita la configuración de este domino como *relay* para habilitar las comunicaciones entre los diferentes dominios que representan a las diferentes líneas o plantas de producción.

Antes de finalizar este capítulo en el que se ha descrito la validación del Meta-OS desarrollado durante esta tesis doctoral en diferentes casos de uso siguiendo un criterio ascendente del nivel de TRL, es conveniente realizar una pequeña reflexión conclusiva, especialmente sobre los casos de uso descritos en esta última sección, ya que tienen lugar en escenarios con un nivel 6 de TRL.

Los casos de uso descritos aún siguen con su desarrollo y constante evolución, puesto que su finalización está prevista para el término del presente año 2025. No obstante, aunque su ejecución no haya finalizado por completo, se ha podido observar el funcionamiento e implementación de las tecnologías desarrolladas durante esta tesis en entornos relevantes pertenecientes a empresas líderes en diferentes sectores verticales (telecomunicaciones, logística portuaria y líneas de producción industriales), en los que el entorno y las condiciones de ejecución son completamente reales y no simuladas para que encajen con las características del sistema desarrollado (TRL 6). Esto se traduce en el manejo de datos reales pertenecientes a diferentes contextos, en condiciones de red que no pueden modificarse a conveniencia (por ejemplo, para evitar restricciones de seguridad) y en la presencia de infraestructura no reemplazable o modificable, tanto de computación como específicas de los casos de uso.

Para finalizar, las tecnologías mencionadas que han podido validarse son parte de los bloques esenciales que habilitan la creación de un Meta-OS para la correcta gestión del continuo *Cloud-Edge-IoT*, esto es, la monitorización del continuo y su modelación utilizando el modelo de datos creado, el acceso ubicuo a la información de contexto, la conexión y federación de dominios, el despliegue de servicios de usuarios mediante el sistema de orquestación (HLO y LLOs para K8s, Docker y containerd) y la plataforma de gestión unificada.

Capítulo 7

Prospectiva

Una vez descrita con un amplio nivel de detalles la implementación de la solución propuesta por esta tesis, es decir, los bloques fundamentales que habilitan la creación de un Meta-OS para gobernar el continuo de computación *Cloud-Edge-IoT*, y su posterior validación en un variado rango de escenarios de validación y casos de uso reales con complejidad incremental, es momento de abordar la parte final o conclusiva de esta tesis doctoral. El doctorando ha decidido incluir un capítulo adicional en esta última parte de la tesis, que típicamente se compone de las conclusiones generales extraídas por el doctorando después de la realización de la tesis y la enumeración de los posibles trabajos futuros o líneas de investigación que pueden emanar de esta. También ha contribuido a dicha decisión el período de tiempo adicional con el que se decidió prorrogar a esta tesis, contribuyendo a la elaboración de un trabajo más completo y en la línea de las tendencias de investigación actuales. El doctorando ha decidido titular este capítulo como "Prospectiva", puesto que, de acuerdo con el diccionario de la RAE, esta palabra tiene como significado "el estudio de las posibilidades o condiciones futuras en una determinada materia".

La intención del doctorando con la inclusión de este capítulo es tratar de establecer una unión entre la investigación e implementación realizada durante la tesis doctoral y las tendencias pujantes en el ámbito de la investigación y de la industria de las TIC. En primer lugar, se va a explorar la inclusión en el Meta-OS de servicios virtualizados utilizando una tecnología alternativa a los contenedores: WebAssembly, ya que durante la investigación acerca del estado del arte fue identificada como la técnica de virtualización más prometedora de las existentes (ver sección 2.7.2), dejando de lado los contenedores. Asimismo, se detallará el gran abanico de posibilidades que permitiría abrir la inclusión de esta tecnología en todos los niveles del Meta-OS. Por último, se van a abordar las líneas futuras de investigación que podrían emanar de esta tesis, empezando desde un punto teórico, pero poniendo especial énfasis en la propuesta de una serie de implementaciones previsiblemente factibles en el Meta-OS desarrollado como resultado de la tesis.

Finalmente, con la escritura de este capítulo de prospectiva se pretende dotar al capítulo final de un espíritu puramente reflexivo y crítico, dando pie a que el doctorando pudiera iniciar un proceso de análisis y reflexión detallado sobre el trabajo realizado durante el desarrollo y escritura de la tesis doctoral para plasmarlo en el capítulo conclusivo.

7.1. Virtualización alternativa y *hardware* de código abierto

La unidad mínima de despliegue del Meta-OS desarrollado en esta tesis doctoral, tanto para los elementos claves que conforman los bloques fundamentales del Meta-OS como para los servicios de usuarios desplegados en la infraestructura del continuo, se estableció en contenedores, tal y como se justificó en el punto 4.2. Esta decisión se sustenta sobre una gran base tecnológica, descrita a lo largo de la sección 2.6, ya que el uso de contenedores como técnica de virtualización se ha convertido en un estándar *cloud-native*.

Sin embargo, como también se especificó en la investigación del estado tecnológico actual, los contenedores no pueden establecerse como la tecnología final y definitiva de virtualización o empaquetado de aplicaciones en este contexto de las TIC, en el que el cambio y la evolución continua son una característica intrínseca. Por lo tanto, también se exploraron alternativas pujantes a la virtualización en contenedores en un subapartado exclusivamente dedicado a este propósito (punto 2.7), entre las que destaca claramente WebAssembly (Wasm, ejecutado en el lado del servidor y no en los navegadores web como fue diseñado inicialmente, descrito en el punto 2.7.2).

En el apartado 2.7.2.1 del estado del arte de esta tesis doctoral describió con detalle el funcionamiento, así como el estado actual y el entusiasmo que ha provocado en la comunidad tecnológica el lanzamiento de WASI 0.2 (o WASI Preview 2), en la que se adoptaba oficialmente el Wasm Component Model para

conseguir el enlazamiento dinámico de módulos Wasm en tiempo de ejecución. Es por ese motivo, y gracias a los avances que se han producido en este último año como consecuencia de dicho lanzamiento y su posterior adopción en *runtimes* y herramientas de compilación en diversos lenguajes de programación, que el doctorando decidió finalmente incluir una primera adaptación del Meta-OS para la integración del despliegue de módulos de Wasm.

Desafortunadamente este entusiasmo aún no ha sido trasladado a los entornos de producción o casos de uso reales, en los que los contenedores siguen siendo el estándar de virtualización *de facto*. Además, aunque Kubernetes (gestionado por la misma CNCF) ha confirmado su prevalencia entre las herramientas de orquestación de contenedores, afirmación que se puede confirmar solamente con observar el hecho de que la mayoría de los proveedores de servicios en la nube ha optado por desarrollar su propia distribución de K8s en lugar de una herramienta alternativa para competir con K8s, la mayoría de esfuerzos a nivel de infraestructura profesional se están realizando en realizar una migración desde entornos tradicionales basados en la creación de VMs específicas para cada servicio o aplicación (en las que a su vez se despliegan contenedores), hasta la creación de entornos puramente *cloud-native* basados en el despliegue de clústeres de K8s.

Además, mientras que avanza el desarrollo de la estandarización de WASI, el nivel de implementación real de Wasm en los lenguajes de programación dista de ser estable y productivo, variando su implementación entre los diferentes lenguajes y los Wasm *runtimes*. Es más, esta compilación no es tan directa ni su proceso está tan estandarizado como el empaquetado de *software* en imágenes de contenedor, necesitando el uso de varias herramientas intermedias (como *wit-bindgens*) y se complica aún más si se opta por seguir el Component Model e intentar la unión de varios módulos. Por todos estos motivos se descartó finalmente la idea inicial del doctorando de compilar en forma de un módulo Wasm alguno de los módulos esenciales del Meta-OS descritos durante el capítulo 5, invirtiendo ese tiempo en la mejora y ajuste de los mismos, y por lo tanto posponiéndola para el futuro, enmarcándola dentro de las líneas futuras de investigación de la tesis, aunque esté justificada a nivel teórico. En su lugar, tal y como se ha afirmado previamente, se optó por realizar una primera integración del despliegue de módulos de Wasm previamente desarrollados por expertos en la materia y cuyo funcionamiento estuviera garantizado, ya que desde el lanzamiento de WASI 0.2.1 estos módulos pueden empaquetarse de forma nativa en imágenes OCI, el mismo estándar que siguen las imágenes de los contenedores.

Siguiendo esta línea de actuación, una de las mayores ventajas que aportan los módulos Wasm es que pueden ejecutarse en cualquier nodo de computación con independencia de la arquitectura de su procesador, siempre y cuando cuenten con un Wasm *runtime* (elemento encargado de la ejecución de estos módulos, equivalente al *container runtime* de bajo nivel) compatible esta arquitectura de CPU del nodo de computación. El uso de módulos Wasm para el despliegue de los elementos básicos del Meta-OS abriría un gran abanico de ventajas y posibilidades,

como la eliminación de la necesidad de compilar imágenes de contenedores específicas para cada arquitectura de CPU, y, por ende, la compatibilidad por defecto con cualquier tipo de nodo de computación, habilitando su integración completa en el continuo *Cloud-Edge-IoT*. Es más, dada la compatibilidad entre contenedores y Wasm, y su probada coexistencia (ver apartado 2.7.3), sumado a que ambas son empaquetadas utilizando el mismo estándar (OCI), con la configuración adecuada del Meta-OS se avanzaría hacia una abstracción total en cuanto a la técnica de virtualización utilizada, tanto para el despliegue de los componentes básicos como de los servicios de usuario.

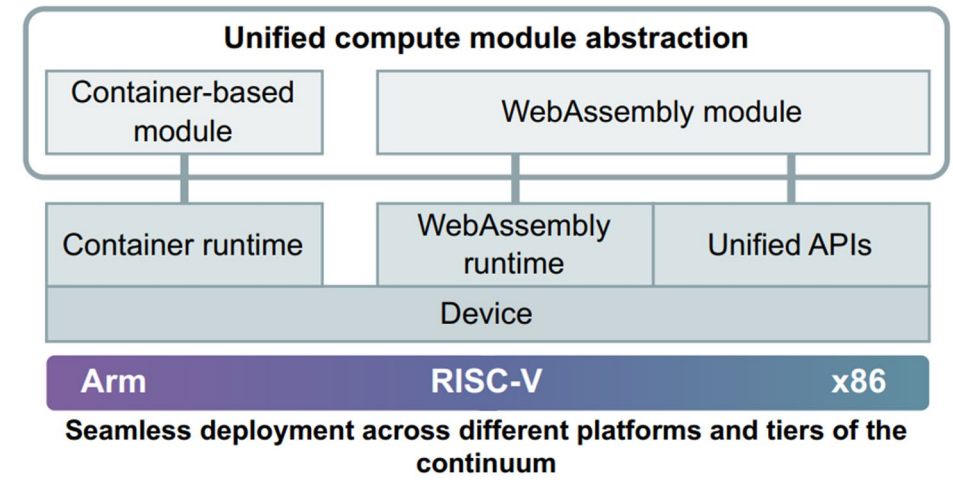


Figura 7.1: Abstracción de la técnica de virtualización en los nodos del continuo

Esta problemática relacionada con la heterogeneidad de arquitecturas de CPU de los nodos de computación se ha intentado abordar con el empaquetado de los componentes básicos en imágenes multi arquitectura que incluyen las arquitecturas más típicas: AMD64 y ARM64. Además, de manera específica, se ha compilado durante esta tesis alguno de estos componentes para la arquitectura de código abierto RISC-V, como, por ejemplo, el LLO de containerd. Sin embargo, este no deja de ser un paso totalmente preliminar que queda bastante lejos de su integración total, ya que estamos hablando de unos pocos componentes que se benefician de la flexibilidad innata de Go en cuanto a la compilación multi arquitectura, aunque también es verdad que actualmente la compilación a módulos Wasm también tiene una fuerte dependencia del lenguaje de programación utilizado, ya que su integración dista de ser plena en todos los lenguajes, como se ha comentado previamente.

Un trabajo similar fue realizado en [332], cuyos autores realizaron una compilación propia de la parte *edge* de KubeEdge (junto con su EdgeMesh) para testear esta tecnología en nodos RISC-V, también gracias a la anteriormente citada flexibilidad de Go. Además, necesitaron construir la imagen de contenedor *pause* de K8s (el componente esencial que habilita la construcción de los *Pods*) para la arquitectura RISC-V, puesto que, como K8s no está disponible de manera oficial para esta arquitectura, la imagen *pause* tampoco lo está. Este trabajo también

incluye un estudio del rendimiento de la ejecución de contenedores que sirve para demostrar que el uso de contenedores en nodos de computación cuyo procesador está basado en RISC-V es totalmente factible, teniendo un rendimiento similar al de nodos basados en ARM.

Uno de los mayores puntos a considerar es la inclusión de la anteriormente citada arquitectura RISC-V, puesto que, a diferencia de las más implantadas arquitecturas x86_64 y ARM, su especificación es de código abierto (fue inicialmente desarrollada por la Universidad de California en 2010 con propósitos educacionales), permitiendo la creación de procesadores sin ningún control por parte de la empresa propietaria de la tecnología, como sucede en el mercado de procesadores x86_64 controlado por las empresas estadounidenses AMD e Intel. La implantación de RISC-V supondría un paso de gigante para avanzar hacia la tan anhelada unión del *software* y del *hardware* bajo licencias de código abierto, ya que permitiría avanzar hacia la creación de un ecosistema libre y colaborativo en el que se primara la innovación y la adaptación específica a los requerimientos de cada caso de uso, puesto que no sería necesario el pago de licencias en ninguno de sus niveles, de manera que las grandes empresas tecnológicas (mayoritariamente estadounidenses) perderían gran parte de su capacidad de influencia y control sobre el ecosistema de las TIC.

Este hecho no se circunscribe únicamente al aspecto tecnológico, ya que entidades gubernamentales de algunas potencias mundiales han puesto su foco en él y lo han incluido en su hoja de ruta estratégica. La Comisión Europea emitió un informe en septiembre de 2022 en el que se recomendaba una hoja de ruta para el avance en la producción y el uso de tanto *software* como *hardware* de licencia abierta para avanzar hacia la autonomía tecnológica de la Unión Europea [333]. La implementación de esta hoja de ruta se puede observar en la financiación de proyectos de investigación como European Processor Initiative (EPI, *Framework Partnership Agreement* —FPA— n.º 101036168 financiado con casi 150 millones de euros) [334] y DARE —Digital Autonomy with RISC-V in Europe— Special *Grant Agreement* n.º 101202459 financiado con 240 millones de euros) [335], cuyo principal propósito es el desarrollo de CPUs competitivos con arquitecturas RISC-V. Además, todo indica que el gobierno chino está cerca de lanzar una política estatal para acelerar el desarrollo y uso de chips basados en RISC-V con el objetivo de esquivar las restricciones impuestas por el gobierno de los Estados Unidos para el acceso a los procesadores de última generación producidos por empresas líderes como Intel o NVIDIA [336].

Esta tendencia pujante de RISC-V también se puede observar al nivel de implantación tecnológica, ya que algunos de los proyectos insignia del ecosistema *cloud-native* de la CNCF ofrecen una versión oficial para RISC-V, como Helm, CoreDNS u el propio containerd. Por otra parte, los Wasm *runtimes* más populares (Wasmer, Wasmtime y WasmEdge), también ofrecen una versión para ser instalada en equipos cuyos procesadores esta arquitectura de CPU.

Finalmente, de forma paralela al desarrollo del Meta-OS resultante de esta tesis, el doctorando realizó unas investigaciones tomando como base la tecnología WebAssembly que dieron como resultado dos artículos de congreso. Estos trabajos también se enmarcan en el contexto del continuo de computación, aunque no están directamente relacionados con el Meta-OS, aunque sus frutos podrían dar lugar a una integración futura. Mientras el primero presenta una propuesta de arquitectura para el despliegue e interacción coordinada y uniforme de módulos Wasm en el continuo permitiendo establecer un flujo de datos coordinado entre ellos (con la ayuda de Orion-LD para el almacenamiento y gestión de sus metadatos, en convivencia con contenedores, el segundo introduce una propuesta dirigida a la solución de los desafíos presentes en el desarrollo de *software*, con la ayuda de una serie de módulos de IA coordinados por unos elementos ejecutados en forma de componentes Wasm.

Inclusión de módulos Wasm en el sistema de orquestación del Meta-OS

La primera decisión de diseño que se tomó para desarrollar la inclusión de módulos Wasm en el sistema de orquestación y despliegue de servicios del Meta-OS consistió en limitar el despliegue de estos módulos a nodos de computación del continuo pertenecientes a clústeres de Kubernetes. Esta decisión se fundamenta en que la ejecución de módulos Wasm en un clúster de K8s puede lograrse haciendo uso de los artefactos y mecanismos que contiene K8s por defecto, proceso que se describió brevemente tanto en el apartado 2.7.3 (ilustrado en la Figura 2.17), como en el artículo de investigación escrito por el doctorando [5]. Como consecuencia, es necesario modificar el LLO de K8s, es decir, el Operador de K8s desarrollado, cuyo funcionamiento fue detallado en el apartado 5.4.4.1).

De tomar la decisión contraria e intentar habilitar cualquier nodo del continuo para la ejecución de módulos Wasm, se necesitaría elegir un único Wasm *runtime* para ser instalado por defecto en todos los nodos que se quiera habilitar para este propósito, lo que se traduciría en una dependencia excesiva en un entorno cambiante y en constante evolución, a diferencia de los contenedores. Además, siguiendo con la arquitectura de los diferentes tipos de LLO no K8s (compuesta por un Operador de K8s, un *broker* NATS y un controlador, ver 5.4.4), conllevaría el desarrollo de un controlador específico para el *runtime* elegido.

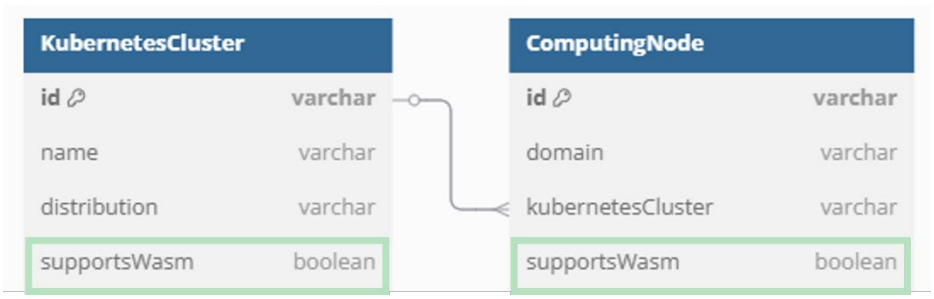


Figura 7.2: Adaptación del modelo de datos para la integración de módulos Wasm

En primer lugar, se realizó una modificación del modelo de datos del continuo (descrito en el punto 5.2.1), concretamente de las entidades *ComputingNode* y *KubernetesCluster*, para indicar el soporte para la ejecución de componentes Wasm tanto a nivel de clúster (indicando si la versión del LLO de K8s cuenta con soporte para Wasm o si esta funcionalidad se encuentra activada) como de nodo. Esta modificación se ilustra en la anterior Figura 7.2.

Para el soporte de esta funcionalidad a nivel del nodo, es necesaria la instalación previa de un Wasm *engine* (Wasmer, Wasmtime o WasmEdge) en el nodo y la correcta configuración del *runtime* de contenedores de alto nivel utilizado. En el caso de CRI-O, es necesario el uso de una versión posterior a la 1.31 para que utilice crun (con soporte nativo para interactuar con los tres Wasm *runtimes* antes mencionados) como *runtime* de contenedores de bajo nivel. Mientras que, si se opta por utilizar containerd, es necesaria la instalación manual del *shim* de containerd correspondiente con el Wasm *runtime* desplegado en el nodo (incluido dentro del proyecto de la herramienta *runwasi*, introducida en el punto 2.7.3) y la posterior configuración de containerd para habilitar el uso de este *shim*.

Gracias a esta actualización del modelo de datos, el HLO (concretamente su elemento Agregador de datos) es capaz de realizar un filtrado previo de los nodos de computación candidatos cuando un usuario requiera el despliegue de un componente que corresponda con un módulo Wasm en lugar de un contenedor. Esta adaptación no habría sido posible sin la previa adaptación del HLO a los requerimientos específicos de este Meta-OS (ya que el doctorando tenía en mente la posible inclusión de componentes Wasm en ese momento), tal y como se describió en el punto 5.4.2).

En cuanto al funcionamiento del LLO de K8s, en primer lugar, crea una suscripción en el *context broker* de su dominio (siempre y cuando no exista) que aplica a las entidades NGSI-LD del tipo *ComputingNode*, para recibir una notificación en el caso de se habilite la ejecución de componentes Wasm (atributo *supportsWasm* con valor *true*) y reaccionar mediante la creación de una nueva *label* en el nodo de K8s con valor *meta-os.eu/wasm=true*. Seguidamente, crea una *RuntimeClass* en el clúster con la configuración adecuada para permitir a K8s desplegar componentes Wasm en los nodos en los que es posible.

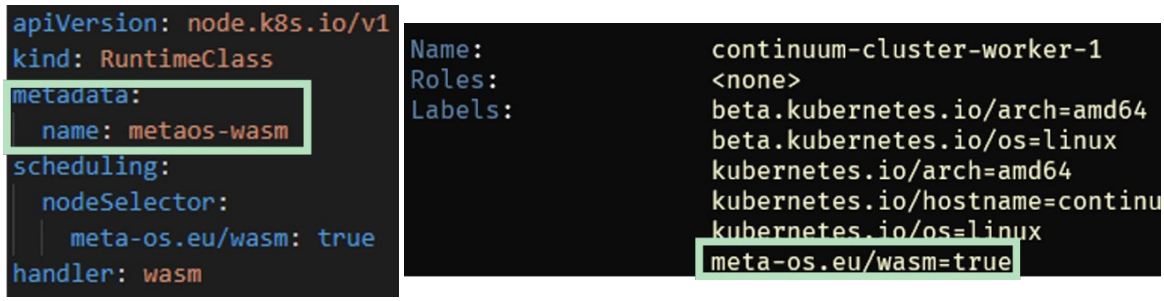


Figura 7.3: *RuntimeClass* creada por el LLO de K8s para la ejecución de módulos Wasm y *labels* del nodo del clúster K8s

Para la correcta integración de esta funcionalidad en el LLO de K8s, ha sido necesaria la adición de un nuevo atributo al *Custom Resource ServiceComponentK8s* (*isWasmComponent: true*) para indicar que el componente del servicio que se va a desplegar corresponde con un módulo Wasm en lugar de un contenedor. Cuando el CR incluye este atributo, el Operador del LLO añade al *spec* del Deployment correspondiente al componente un nuevo atributo (*runtimeClassName: metaos-wasm*) para indicar a la API de K8s que el *Pod* correspondiente al *Deployment* ha de ser ejecutado utilizado la *RuntimeClass* anteriormente configurada, es decir, por el Wasm *runtime* en lugar del *container runtime* de bajo nivel. Por último, es necesario mencionar que se ha decidido excluir los componentes Wasm del sistema de conexión de servicios y componentes basado en WireGuard (detallado en el punto 5.4.2.2).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/created-by: urn_ngsi-ld_LowLevelOrchestrator_Cluster01_Kubernetes
    app.kubernetes.io/instance: urn_ngsi-ld_Service_e0787d3d7ba4_Component_first
    app.kubernetes.io/managed-by: metaos-project.eu
    app.kubernetes.io/name: metaos-service-e0787d3d7ba4-component-first
    app.kubernetes.io/part-of: urn_ngsi-ld_Service_e0787d3d7ba4
  name: metaos-service-e0787d3d7ba4-component-first
  namespace: default
spec:
  runtimeClassName: metaos-wasm
  template:
    spec:
      nodeSelector:
        kubernetes.io/hostname: continuum-cluster-worker-1
      containers:
```

Figura 7.4: Uso de la *RuntimeClass* específica para la ejecución de componentes Wasm en el *Deployment* creado por el LLO de K8s

7.2.Líneas futuras de investigación

En esta segunda parte del capítulo de prospectiva, se van a abordar una serie de posibles líneas futuras de investigación o trabajos de implementación aplicados que emanan de la realización de esta tesis doctoral. Estas líneas de investigación o trabajos futuros pueden dividirse en tres bloques diferenciados con base al tipo de actuación en la que se circunscriben.

Antes de proceder con estos tres bloques, una acción a futuro lógica y transversal consistiría en la validación del Meta-OS desarrollado en unos verticales diferentes, como, por ejemplo, la agricultura. Este escenario supondría la validación de este Meta-OS en escenarios con dominios *edge* distribuidos (correspondientes con los mismos campos agrícolas), escenarios que tradicionalmente no han estado

preparados para la ejecución de tecnologías TIC, aunque en la última década se han empezado a implantar sistemas basados en el IoT (activación del riego, monitorización de las condiciones ambientales, control de plagas, etc.). Otro escenario de validación interesante consistiría en el despliegue del Meta-OS en entornos de ciudades inteligentes, un vertical típico en el ámbito del IoT, en los que la división de dominios correspondería con las diferentes ciudades o incluso con los barrios de las mismas para una mayor granularidad.

Colaboración con otros proyectos de investigación

El Meta-OS desarrollado durante esta tesis doctoral no se trata del único Meta-OS de su naturaleza que se encuentra en fase de desarrollo. Este hecho se ha podido comprobar en la sección 3.4, durante la cual se describieron los proyectos de investigación dentro de la acción de la EC HORIZON-CL4-2021-DATA-01-05 para abordar la creación de Meta-OS para gestionar el continuo [251], ahondando con mayor detalle en FLUIDOS, ICOS y aerOS. Por lo tanto, es lógico pensar que uno de los trabajos futuros claves para avanzar en la construcción de un Meta-OS más sólido es establecer una sinergia mediante la integración de las funcionalidades claves que aporta cada uno de ellos. Por ejemplo, mientras que este Meta-OS pone su foco en la monitorización de los nodos, la federación entre dominios y la orquestación de servicios (es decir, los bloques fundamentales definidos en el punto 4.5), influyendo directamente al proyecto aerOS (en el que recordemos, ha participado de manera activa el doctorando), este proyecto implementa una funcionalidad para la definición de un *data fabric* de manera federada entre los dominios del continuo, habilitando la creación de productos y flujos de datos entre servicios desplegados en el continuo. Asimismo, sería interesante la integración del sistema de recursos proporcionado por FLUIDOS mediante su protocolo REAR. La colaboración entre los impulsores de los diferentes Meta-OS permitiría avanzar hacia la creación de un único Meta-OS con tecnología europea para apostar claramente por el liderazgo en el contexto del continuo *Cloud-Edge-IoT*.

De igual manera, se abre la puerta a la utilización del Meta-OS en otros proyectos o iniciativas de investigación con diferentes propósitos, pero cuya acción se desarrolla en el continuo de computación, facilitando la gestión del mismo para que puedan enfocarse en sus objetivos específicos. Este es el caso del proyecto HE SAFE-6G (*A Smart and Adaptive Framework for Enhancing Trust in 6G Networks*, GA n.º 101139031) [337], en el que se están empezando a utilizar bloques básicos del Meta-OS desarrollado durante esta tesis debido a que el grupo SATRD forma parte de su consorcio. Para este propósito, los desarrolladores del proyecto SAFE-6G han decidido adaptar el LLO de K8s para que permita el despliegue de Helm charts con una configuración personalizada, ya que Kubernetes se trata del único sistema de gestión de contenedores utilizado en los nodos del continuo al enfocarse en infraestructura de proveedores de servicios de telecomunicaciones para avanzar hacia el despliegue de redes 6G en el continuo.

En la línea de esta primera reflexión, la sinergia entre varios de estos proyectos será garantizada a través de la participación del grupo SATRD (y, por tanto, del doctorando) en un proyecto de investigación recientemente comenzado denominado O-CEI (*Open CloudEdgeIoT Platform Uptake in Large Scale Cross-Domain Pilots*, GA n.º 101189589) [338]. Este proyecto reúne a 58 socios de 19 países, entre los que se encuentran destacados agentes industriales, organizaciones de investigación, instituciones académicas, pymes y organizaciones sin ánimo de lucro. El objetivo del proyecto es crear un ecosistema orientado a fomentar el mercado de "utilidades" (tecnologías) del entorno *Cloud-Edge-IoT* (CEI) centradas en avanzar hacia una mayor flexibilidad energética. Dentro de este proyecto las tecnologías del Meta-OS de aerOS (entre las que se encuentran los elementos esenciales del Meta-OS de esta tesis) se integrarán con un *marketplace* de una forma abierta. Además, los proyectos FLUIDOS, ICOS y NebulOuS también estarán representados, tratando de integrar sus tecnologías bajo unas premisas de innovación abierta y accesible.

En este sentido, O-CEI pretende crear una plataforma CEI abierta que haga hincapié en la interoperabilidad, la seguridad y la fiabilidad. La plataforma pretende construir un ecosistema para acelerar la innovación y la adopción de soluciones CEI, permitir la flexibilidad energética, establecer normas CEI y crear modelos de negocio viables y escalables. Por tanto, los desarrollos de esta tesis doctoral se verán beneficiados de la existencia de este proyecto por las siguientes razones: (i) utilización a gran escala de las herramientas de orquestación, federación y despliegue de microservicios en entornos *Cloud-Edge-IoT* de empresas reales, (ii) contribuir a un objetivo de Sostenibilidad como un mejor y más eficiente uso de los recursos energéticos, (iii) potencial inclusión en un estándar abierto europeo, lo cual situaría los avances de esta tesis como una referencia en los despliegues en los próximos años.

Mejora directa de las actuales funcionalidades soportadas por el Meta-OS

El establecimiento de canales de comunicaciones entre los componentes pertenecientes a un mismo o a diferentes servicios ha supuesto uno de los mayores desafíos en el desarrollo e implementación del sistema de orquestación de servicios del Meta-OS, puesto que en el escenario más complicado estos componentes se encontrarían totalmente distribuidos en el continuo, es decir, desplegados en distintos nodos de computación pertenecientes a diferentes dominios, que a su vez están conectados a diversas redes.

Aunque la primera implementación para la conexión de diferentes componentes desplegados por el Meta-OS se ha realizado tomando WireGuard como base tecnológica, el siguiente paso se fundamentaría en la construcción de una nueva implementación basándose en la herramienta *libp2p*, que se utilizó para establecer las comunicaciones entre dominios para habilitar la federación entre ellos. Este nuevo enfoque consistiría en el despliegue de un nuevo elemento en cada nodo

del continuo para que los servicios desplegados en el mismo fueran capaces de enviar sus datos a través de los canales de comunicaciones P2P establecidos entre ellos. También sería interesante explorar la creación de una red de comunicaciones por dominio, en la que participaran todos los nodos pertenecientes a este, haciendo uso de los mecanismos de autodescubrimiento proporcionados por *libp2p*. Sin lugar a duda, se trata de un desafío muy interesante que constituye una de las líneas de investigación futuras con un mayor potencial, pero a su vez difícil de solucionar de una manera eficiente.

Por otro lado, cuando la especificación de WASI alcance un estado estable y de lugar a un entorno de desarrollo y ejecución de componentes Wasm lo suficientemente maduro para ser utilizado en entornos con al menos un nivel de TRL 6, sería necesario explorar la compilación de los elementos que componen los bloques fundamentales del Meta-OS, además de mejorar la integración de componentes Wasm en el sistema de orquestación del Meta-OS.

Como se ha descrito en el punto 5.7, actualmente se están llevando a cabo algunas tesis doctorales enfocadas en el estudio e implementación de sistemas de seguridad distribuidos en el ámbito del continuo de computación. Cuando estas acciones cuenten con un grado de madurez suficiente, se podría estudiar la integración del Meta-OS con las soluciones desarrolladas en ellas. Asimismo, el Meta-OS podría servir como un potencial banco de pruebas para validarlas.

Finalmente, se podría mejorar el nivel de observabilidad durante el tiempo de ejecución de todos los componentes desarrollados que conforman los bloques fundamentales del Meta-OS mediante la inclusión de un sistema de monitorización y observabilidad basado en la herramienta de la CNCF OpenTelemetry [339], ya que puede integrarse con Prometheus y también enlazaría con el sistema de notificaciones del Meta-OS. Además, sería interesante retomar el camino iniciado para habilitar la monitorización del consumo de energía del continuo, tanto a nivel de los nodos de computación como de los servicios desplegados en ellos.

Extensión del Meta-OS mediante la implementación de funcionalidades auxiliares

Mientras que en el anterior bloque se han detallado un par de mejoras clave de funcionalidades soportadas por los bloques esenciales del Meta-OS actual, una línea de investigación futura que se atisba interesante consiste en la identificación de funcionalidades auxiliares que serían interesantes que soportara el Meta-OS para su posterior construcción sobre los bloques fundamentales del Meta-OS.

Una materia para explorar se trataría de la expansión del Meta-OS mediante la integración de un sistema para el entrenamiento y la inferencia colaborativa de modelos de IA, ya que permitiría aprovechar el entorno de naturaleza distribuida y descentralizada del continuo de computación. Este mismo sistema podría ser utilizado para mejorar los propios algoritmos de IA intrínsecos al Meta-OS, como se trata del algoritmo utilizado por el elemento (Re)Allocator del HLO para la

elección del nodo de computación de despliegue de un componente de un servicio, así como para el desarrollo de un modelo encargado de la detección de posibles anomalías en el correcto funcionamiento del Meta-OS.

Por otra parte, sería interesante explorar la creación de un sistema para el despliegue de flujos de ejecución de tareas basado en el despliegue y la ejecución de componentes (en cualquier nodo del continuo) siguiendo un orden previamente definido, en el que el resultado de cada tarea pudiera ser reutilizado como datos de entrada para la ejecución de la siguiente. Este sistema se construiría sobre el actual sistema de orquestación descentralizada de servicios. Además, se podrían utilizar los diferentes *context brokers* de los dominios del continuo para almacenar los metadatos relacionados con la ejecución de las diferentes tareas, e incluso directamente ciertos resultados en la medida de lo posible.

Capítulo 8

Conclusiones

El trabajo realizado durante esta tesis doctoral ha comenzado explorando la transición de los paradigmas de computación utilizando como hilo conductor los proyectos de investigación en los que ha participado el doctorando. Este recorrido parte del *cloud computing*, aborda la aparición del *edge computing* y culmina en el continuo de computación *Cloud-Edge-IoT*, en el que se intentan integrar ambos paradigmas para englobar el espectro de computación al completo.

Tras este análisis, el doctorando ha intentado abordar la problemática relacionada con la no existencia de un sistema para gestionar eficientemente la heterogeneidad intrínseca al continuo mediante el desarrollo de los bloques básicos que habilitan la creación de un Meta-OS para gobernar este entorno distribuido. El desarrollo de este Meta-OS se ha realizado utilizando las tecnologías punteras que se han descubierto y analizado después del extensivo estudio inicial del estado del arte tecnológico. Además, este Meta-OS ha sido validado en diferentes casos de uso con un nivel de TRL ascendente, incluyendo casos de uso reales. Finalmente, el sistema desarrollado ha sido preparado para su integración con tecnologías de virtualización innovadoras, dando como fruto una reflexión final sobre las acciones

de investigación futuras y las posibles mejoras que podrían implementarse en trabajos posteriores.

Después de describir en el capítulo anterior la prospectiva del trabajo de investigación e implementación realizado durante esta tesis, este capítulo final incluye una serie de conclusiones obtenidas en cada uno de los capítulos de la misma. Además, como colofón, el doctorando ha decidido incorporar una serie de reflexiones finales sobre la innovación aportada por esta tesis doctoral.

8.1.Conclusiones finales

En este primer subapartado se van a exponer las principales conclusiones que se han obtenido después de la escritura de cada capítulo que compone esta tesis doctoral, así como de la realización de los trabajos asociados. Es necesario apuntar que este contenido no se trata de un simple resumen de las acciones desempeñadas, sino que el doctorando pretende dotar a este contenido de una cierta cohesión y de un espíritu reflexivo en la que se plasmen los aspectos más relevantes. En consecuencia, se pretende que este contenido pueda servir a un lector interesado en la materia para identificar los aspectos de su propio interés, y así poder saltar directamente a los capítulos relacionados para estudiarlos más detalladamente.

Estado del arte tecnológico

- La aparición del *edge computing* permitió la migración progresiva de cargas de computación a dispositivos locales o más cerca de la fuente de datos (como los dispositivos IoT), de manera que se consigue reducir latencia en las comunicaciones y aumentar la seguridad e integridad de los datos, al mismo tiempo que permite avanzar hacia una mayor eficiencia energética. Sin embargo, como contrapartida los dispositivos de computación del *edge* cuentan con capacidades más limitadas y una mayor heterogeneidad.
- Existe una clara tendencia para la transferencia de tecnologías, herramientas y patrones de diseño propios del *cloud computing* (*cloud-native*) al *edge computing* (*edge-native*). Esta tendencia se puede observar en el lanzamiento de proyectos e iniciativas por parte de entidades relevantes como la CNCF, Eclipse Foundation o la Comisión Europea. Además, los principales proveedores de servicios de computación en la nube ofrecen soluciones propias del *edge computing* como una extensión de sus plataformas *cloud*.
- El continuo de computación *Cloud-Edge-IoT* pretende englobar todos los elementos de computación (nodos, redes, aplicaciones...) desplegados a lo largo de todos segmentos del espectro de computación (*cloud, edge, far-edge,* IoT...) con el propósito de alcanzar una relación de colaboración fluida entre ellos y dotar de una capa de uniformidad a un entorno ya de por si altamente heterogéneo.

- Los *context brokers* han tenido un papel fundamental en las plataformas IoT para gestionar la información de contexto aportada por los sensores y actuadores. Recientemente se ha explorado su potencial para la monitorización de entornos de computación alternativos (entre los que encajaría el continuo *Cloud-Edge-IoT*) mediante el uso de un modelo de datos adecuado. Esta acción se plasma en la extensión del estándar NGSIv2 a NGSI-LD con la inclusión de elementos semánticos y mecanismos para la distribución de la información de contexto entre múltiples instancias de CBs.
- El uso de comunicaciones asíncronas (principalmente a través de *brokers* de mensajería) encaja adecuadamente en el ámbito del *edge computing*, puesto que permiten un desacoplamiento temporal en un entorno en el que las conexiones tienen una baja fiabilidad. Asimismo, se han desarrollado *brokers* con características específicas (federación, consumo reducido de recursos, baja latencia, etc.) para ser desplegados en este tipo de entornos, como NATS, en contraposición a los tradicionales, como Kafka.
- La conectividad entre servicios desplegados en entornos del *edge computing* se presenta como uno de los grandes desafíos a resolver debido a la gran heterogeneidad (topología, protocolos, medios de conexión...) y las condiciones específicas (inestabilidad de las conexiones, uso de LANs privadas y de elementos adicionales de seguridad...) con la que cuentan las diferentes redes utilizadas en estos entornos. Existen diferentes alternativas para intentar abordar estos problemas como el establecimiento de túneles o VPNs modernas (WireGuard), el uso de técnicas de NAT transversal o la creación de redes descentralizadas o distribuidas (*libp2p*).
- Los contenedores se convirtieron en la tecnología de virtualización estándar del patrón de diseño *cloud-native* debido a las ventajas que aportan comparados con las VMs: no necesitan un hipervisor ni un OS dedicado, son más ligeros y escalables, y permiten una ejecución más dinámica.
- Aunque los contenedores son el estándar *cloud-native* para la ejecución de aplicaciones, este paradigma se ha trasladado con éxito al entorno *edge-native*. Este hecho se puede comprobar en el desarrollo de *runtimes* de contenedores específicos para el *edge* tanto de alto nivel (iSulad) como de bajo nivel (crun o youki), OS específicos para la ejecución de contenedores (Balena o Pantacor), distribuciones de K8s específicas (K3s o MicroK8s) y alternativas para la orquestación de contenedores (incluso su adaptación directa como es KubeEdge). Estas alternativas a K8s para el *edge* cuentan con una arquitectura en la que un nodo central o *cloud* ejerce el control sobre los nodos distribuidos o *edge*.

- El paradigma de ejecución de servicios *serverless* puede ser utilizado para que equipos con una capacidad de computación limitada requieran la ejecución de funciones o servicios a equipos más potentes que se sitúan en la capa de arquitectura directamente superior. De esta manera se produce un traspaso efectivo de responsabilidades de manera colaborativa con el propósito de balancear el uso de los recursos de computación en el ámbito del continuo.
- En los últimos años han surgido varias alternativas a los contendores para la virtualización de servicios como microVMs o unikernels, entre las que destaca claramente WebAssembly debido al tamaño reducido del binario de sus módulos, su bajo consumo de memoria RAM, su rápida capacidad de arranque y sus tiempos reducidos de respuesta. Estas características, junto con su independencia de la arquitectura de CPU de la máquina en la que se ejecutan, hacen de Wasm una tecnología ideal para su uso en entornos propios del *edge computing*.
- El lanzamiento de la versión 0.2 de WASI (la interfaz que habilita la ejecución de módulos Wasm fuera de los navegadores web, en las mismas máquinas) supuso un hito en la estandarización de Wasm puesto que sus funcionalidades se consideran estables y se basa oficialmente en el Wasm Component Model. Se ha visto como un paso esencial para su estandarización definitiva, su uso por todo tipo de desarrolladores de *software* y su consiguiente adopción en entornos de producción.
- La coexistencia de contenedores y módulos Wasm está ampliamente validada, como se puede comprobar en el soporte oficial de Wasm por parte de Docker o K8s y el uso de estas como herramientas de despliegue común de ambos tipos de cargas de trabajo.

Transición hacia el continuo Cloud-Edge-IoT

- La participación del doctorando en varios proyectos de investigación desde el año 2016 le ha permitido observar la evolución en los puntos de interés o paradigmas tecnológicos sobre los que se estos construyen. Por lo tanto, se establece un patrón de transición hacia el actual paradigma del continuo de computación *Cloud-Edge-IoT* que empezó con la investigación de las plataformas IoT e interoperabilidad entre diferentes fuentes de información. De forma paralela emergió el *cloud computing*, cuyas técnicas fueron utilizadas para la materialización de estas plataformas IoT. Seguidamente, con la aparición del *edge computing* el foco de interés se movió hacia el estudio y desarrollo de sistemas colaborativos y descentralizados *Cloud-Edge,* que cuentan con fuerte dependencia de elementos centralizados en la nube. Finalmente, aprovechando su fuerte influencia en el ámbito del *edge computing* y

del IoT, la EC centró sus esfuerzos de innovación en el continuo de computación para intentar afianzarse como agente líder en este ámbito.

- Tanto el doctorando como los directores de la tesis fueron los impulsores principales de la definición de los bloques de arquitectura del sistema de orquestación del WG5 de la TF3 de la iniciativa EUCloudEdgeIoT, que se construye alrededor de la federación entre dominios. Esta arquitectura se encuentra actualmente en proceso de estandarización dentro del programa ISO/IEC JTC 1/SC 41.
- La Comisión Europea identificó el desarrollo de un Meta-OS para gobernar o gestionar el continuo de computación como uno de sus objetivos para abordar este desafío técnico. Este hecho se plasma en la financiación de varios proyectos de investigación dentro de la acción HORIZON-CL4-2021-DATA-01-05, entre los que destacan ICOS, FLUIDOS y aerOS.

Diseño de la solución

- La investigación realizada durante los capítulos anteriores permitió al doctorando llegar a la conclusión de que la herramienta tecnológica que mejor encaja para conseguir una implementación real del continuo *Cloud-Edge-IoT* es un Meta-OS para la gestión del mismo. Puesto que el desarrollo de un Meta-OS completo sería un objetivo demasiado ambicioso y poco realista para una tesis doctoral, se decidió centrar los esfuerzos en la materialización de los bloques básicos que habilitan un Meta-OS para este propósito.
- El elemento más granular del continuo para la ejecución de servicios se establece en un nodo de computación. Este nodo tiene que cumplir con una serie de requisitos básicos para ser considerado como tal, entre los que destaca la capacidad para ejecutar contenedores mediante cualquier sistema de orquestación de contenedores (K8s, Docker, containerd, Balena, etc.). Si un elemento de computación no cumple con dichos requisitos, como podría ser el caso de ciertos tipos de dispositivos IoT, este debe ser asignado a otro nodo.
- Los nodos de computación se agrupan en entidades administrativas llamadas dominios, que comparten la misma instancia de los bloques elementales del Meta-OS. Esta agrupación ha de realizarse previamente a la instalación del Meta-OS bajo los criterios de los administradores de la infraestructura, pero siempre de acuerdo con una cierta lógica técnica.
- Los bloques esenciales que habilitan un Meta-OS para gobernar el continuo de computación son cuatro: (i) Federación entre dominios en tres niveles (servicios, información de contexto y orquestación); (ii) Orquestación descentralizada de servicios dividida en dos niveles (alto —HLO— y bajo nivel —LLO—, siguiendo la arquitectura de la TF3

de EUCEI); (iii) Monitorización y observabilidad; y (iv) Gestión unificada del continuo, a través del dominio designado como *entrypoint*.

- Respecto a los dos niveles o bloques del sistema de orquestación, mientras que el HLO tiene un enfoque general para todo el continuo (se despliega una única instancia en cada dominio), el LLO tiene un carácter específico y local que depende del sistema de gestión de contenedores utilizado por un nodo para ejecutar contenedores.

Integración de la solución

- La no existencia de un modelo de datos específico para la modelación del continuo *Cloud-Edge-IoT* hizo necesaria la creación de un modelo de datos desde cero, adaptada a la solución desarrollada, pero con un espíritu generalista para plasmar la realidad del continuo. Este modelo de datos se implementa utilizando el estándar NGSI-LD (en forma de entidades y atributos) para poder ser utilizado en el CB FIWARE Orion-LD, en cuya mejora y adaptación participó activamente el doctorando.
- La monitorización del continuo se materializa mediante el desarrollo de elementos con el propósito de monitorizar sus dos elementos claves y persistir esta información de contexto en el CB de cada dominio: los nodos de computación y los servicios desplegados por los usuarios. Para este desarrollo se han tenido en consideración el uso de herramientas *cloud-native* ampliamente utilizadas como Prometheus y cAdvisor. Finalmente se descartó la monitorización del consumo de energía debido a la dificultad para obtener métricas reales en entornos virtualizados.
- La federación entre dominios del continuo está habilitada por dos características o funcionalidades proporcionadas por el Meta-OS: (i) La ubicuidad en el acceso a la información de contexto global del continuo en tiempo real con independencia del dominio en el que se realice, evitando la replicación de la información; (ii) El establecimiento de canales de comunicaciones entre las instancias de los elementos claves del Meta-OS (HLO, Federador, CB...) desplegadas en cada dominio, que permiten superar los problemas derivados de la heterogeneidad a nivel de red, como el uso de redes privadas.
- La inclusión del elemento Entrypoint balancer permite esquivar la adición de un punto de centralización, puesto que su tarea única consiste en la distribución de las peticiones de orquestación de servicios entre los HLOs de los diferentes dominios que conforman el continuo de computación. Esta distribución obedece a la ejecución de un algoritmo de balanceo de cargas del tipo *Wighted Least Connections*.

- El sistema de orquestación descentralizada de servicios del Meta-OS aporta un mecanismo para la gestión del ciclo de vida completo de los servicios desplegados en tres niveles (Meta-OS, LLO y contenedor). Además, permite la conexión directa de componentes de los servicios con independencia del domino en el que estén desplegados mediante la creación de túneles de WireGuard.
- La implementación del HLO del Meta-OS está fuertemente basada en el modelo de datos del continuo, así como en la federación entre dominios. Las acciones del HLO se realizan de manera asíncrona para cada componente de un servicio, finalizando con la elección de un nodo para su despliegue y con el posterior envío de la orden de ejecución al LLO asociado con dicho nodo.
- Los LLOs del Meta-OS se construyen sobre el *Operator pattern* de K8s, que actúa como solución inclusiva para todos los tipos de sistemas de gestión de contenedores. Mientras que el LLO de K8s realiza las operaciones sobre el mismo clúster de K8s en el que está desplegado, los demás tipos de LLOs necesitan incluir un elemento adicional (el controlador) que se encargue de realizar las acciones demandadas en el nodo de computación, con el que se comunican de manera asíncrona mediante el intercambio de mensajes (modelados con la especificación CloudEvents) a través del *broker* NATS.
- El Meta-OS aporta una plataforma para la gestión del continuo desde un único punto central (el dominio *entrypoint*), que se trata de una aplicación web SPA junto con un *backend* diseñada de manera flexible para que pueda ser migrada entre dominios.
- Se definió una estrategia común de empaquetado basada en la creación de una serie de *labels* para identificar de manera inequívoca a cada elemento desplegado en forma de contenedor. Además, el doctorando decidió que la instalación del Meta-OS estuviera solamente disponible para máquinas Docker y clústeres de K8s. Para facilitar el proceso de instalación en K8s mediante Helm charts, se desarrolló la herramienta Helm chart generator, que puede ser utilizada para empaquetar cualquier tipo de desarrollo y mejorar la adopción de tecnologías *cloud-native*.
- El doctorando decidió no incluir la seguridad como un bloque básico del Meta-OS debido a que se trata de un sujeto de investigación complejo por sí mismo. No obstante, está presente en la implementación del Meta-OS (las comunicaciones entre dominios y componentes están encriptadas, por ejemplo). De la misma manera, los Operadores de los LLOs se integran con el sistema de fiabilidad y reputación del Meta-OS de aerOS, que se pivota sobre la tecnología de DLT IOTA.

Casos de uso

- Los casos de uso para la validación del Meta-OS desarrollado durante la tesis han sido seleccionados siguiendo una estrategia de validación escalonada basada en su nivel de TRL, que incluye los niveles 4, 5 y 6. De esta manera, ha sido posible aumentar la complejidad de estos escenarios progresivamente con la introducción de nuevos requisitos y desafíos.
- Ha sido posible la validación de la escalabilidad de la solución para la federación de dominios propuesta mediante la realización de una serie de pruebas. Asimismo, los resultados de estas pruebas muestran que la federación entre dominios tiene una fuerte dependencia de factores externos al propio Meta-OS que no es posible controlar, entre los que destacan las condiciones de las redes a las que están conectados los nodos que conforman los dominios.
- La participación del doctorando en el proyecto de investigación aerOS ha permitido al candidato acceder a la infraestructura de computación de casos de uso reales para realizar una validación del Meta-OS en escenarios de TRL de nivel 6. Además, ciertos elementos claves del Meta-OS han sido utilizados en la demostración de este proyecto delante de evaluadores oficiales de la Comisión Europea.
- El despliegue y uso del Meta-OS desarrollado en esta tesis ha actuado como elemento habilitador clave de varios casos de uso reales (TRL 6) para la consecución de sus objetivos específicos, que no están directamente relacionados con el propio Meta-OS. Estos casos de uso pertenecen a empresas líderes de diferentes sectores verticales (telecomunicaciones, logística portuaria y líneas de producción industriales), con una gran variedad de infraestructura de computación (centros de datos, dispositivos IoT, máquinas preexistentes heredadas...), infraestructura específica (grúas, máquinas industriales, robótica...), condiciones operativas, fuentes y tipos de datos.

Prospectiva

- La idea inicial de doctorado de compilar algunos componentes de los bloques esenciales del Meta-OS como módulos o componentes Wasm se descartó por la falta de estandarización del proceso junto con su fuerte dependencia con el lenguaje de programación utilizado debido a que aún se encuentra en estado experimental.
- La extensión del Meta-OS para soportar la ejecución de servicios virtualizados utilizando WebAssembly (Wasm) permite abrir una ventana de innovación en el continuo, ya que esta tecnología se establece como una clara alternativa futura a los contenedores, que además comparte su empaquetado en imágenes OCI. Además, debido a que los componentes Wasm pueden ejecutarse en cualquier nodo de

computación con independencia de la arquitectura de su CPU, permite la integración directa de arquitecturas de CPU de código abierto e innovadoras como RISC-V.

- Las líneas futuras de investigación o trabajos futuros se agrupan en tres bloques diferenciados: (i) la colaboración con otros proyectos de investigación para integrar funcionalidades adicionales al Meta-OS o para la utilización directa del Meta-OS para gestionar el continuo de computación; (ii) la mejora directa de las funcionalidades del continuo mediante el desarrollo de una red P2P entre nodos para la conexión de servicios basada en *libp2p*, el desarrollo de un sistema de observabilidad basado en OpenTelemetry y la compilación y ejecución de elementos del Meta-OS en componentes Wasm; (iii) la extensión del Meta-OS mediante la implementación de funcionalidades auxiliares, como un sistema para el entrenamiento y la inferencia colaborativa de modelos de IA, o un sistema para el despliegue de flujos de ejecución de tareas.

8.2.Reflexiones sobre la innovación

El doctorando ha pretendido en todo momento que la solución desarrollada en esta tesis doctoral se construyera sobre las tecnologías, herramientas, paradigmas de computación y líneas de investigación más recientes o vanguardistas, de manera que esta tesis no quedara desfasada al poco tiempo de ser defendida, o incluso durante su misma realización. Este hecho se ve reflejado en la revisión continua del estado del arte, que ha permitido al doctorando ser consciente del rápido avance en el desarrollo y actualización de estas tecnologías. A modo de ejemplo, KubeEdge carecía de una documentación clara para su instalación en el momento en el que se empezó a desarrollar el *software* de los bloques básicos del Meta-OS, pero en el momento de escribir esta reflexión ya ha alcanzado el estado de *Graduated project* de la CNCF para ser considerado el estándar *de facto* para el uso de Kubernetes en entornos del *edge computing*. Asimismo, el Wasm Component Model se lanzó en la fase final de la producción de la tesis (principios de 2024).

En el capítulo anterior de prospectiva se llevó a cabo un esfuerzo adicional para profundizar en esta intención, preparando así el Meta-OS desarrollado para su integración con la tecnología de virtualización más prometedora: WebAssembly. Si bien WebAssembly aún no ha alcanzado un grado de madurez suficiente para su implementación en entornos operativos reales, la investigación llevada a cabo en esta tesis ha permitido consolidar un conocimiento estructurado sobre esta tecnología. Este conocimiento ha quedado plasmado en una sección extensa (2.7.2) cuyo objetivo es proporcionar una documentación coherente y de fácil acceso, en un área en la que la documentación disponible no acaba de ser clara y está fragmentada y dispersa en múltiples fuentes, como especificaciones técnicas, tutoriales técnicos y repositorios de código. En consecuencia, esta documentación puede servir para la iniciación de los lectores de la tesis que no estén familiarizados

con Wasm, o incluso como material de consulta para el propio doctorando en el futuro.

Uno de los aspectos más innovadores de la tesis doctoral radica en la contribución principal a la definición de los bloques de arquitectura del sistema de orquestación del WG5 de la TF3 de la iniciativa EUCloudEdgeIoT (en la que la federación entre dominios tiene un papel fundamental) que está en proceso de estandarización dentro del programa ISO/IEC JTC 1/SC 41. En esta línea, los bloques fundamentales del Meta-OS pueden ser vistos como un ejemplo de la implementación de esta arquitectura que puede ser tomada como referencia, o incluso como una muestra de su validación a nivel técnico.

La creación de un modelo de datos específico para la modelación del continuo de computación *Cloud-Edge-IoT* se trata de otra aportación innovadora a la materia, puesto que hasta el momento no existe un modelo de datos u ontología aplicada para este paradigma de computación. Por ende, se decidió iniciar el proceso para la inclusión formal de este modelo de datos dentro de la iniciativa FIWARE Smart Data Model para que su uso esté disponible de manera abierta por toda la comunidad.

De manera adicional a los sistemas desarrollados por el propio doctorando, la realización de esta tesis también ha contribuido a la mejora de tecnologías o productos de código abierto ya existentes, por lo que se podría afirmar que el doctorando ha contribuido a la mejora del ecosistema tecnológico actual. Esta contribución se puede observar claramente en el caso del *context broker* Orion-LD, ya que el doctorando ha contribuido a su desarrollo mediante la detección de fallos, o la petición y probatura de nuevas funcionalidades. Este proceso puede ser consultado de forma abierta mediante los más de 20 *issues* directos publicados por el doctorando en el del repositorio oficial de GitHub de Orion-LD [340]. En menor medida, este método de interacción mediante *issues* también ha sido utilizado en repositorios de tecnologías como KubeEdge, FIWARE PEP Proxy, Swagger, Kepler o Balena.

Por último, en la Tabla 8.1 que se muestra a continuación, se ha realizado un ejercicio para relacionar las principales publicaciones en las que ha participado el doctorando durante la realización de esta tesis doctoral con los capítulos en los que su influencia ha sido mayor. De esta manera, se pretende validar la innovación en las diferentes asunciones, diseños e implementaciones de la tesis.

Tabla 8.1: Relación de artículos publicados con los capítulos de la tesis

Publicación	Capítulo
Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions	2- Estado del arte tecnológico
	4- Diseño de la solución

Self- Capabilities of Cloud-Edge Nodes: A Research Review*	2- Estado del arte tecnológico
	5.2- Monitorización y observabilidad
Framework and Methodology for Establishing Port-City Policies Based on Real-Time Composite Indicators and IoT: A Practical Use-Case	3.2- Plataformas IoT centralizadas
Tactile Internet in Internet of Things Ecosystems	3.3- Sistemas descentralizados *Cloud-Edge*
A Novel Orchestrator Architecture for Deploying Virtualized Services in Next-Generation IoT Computing Ecosystems	3.3- Sistemas descentralizados *Cloud-Edge*
Novel Approach to Self- Capabilities in IoT Industrial Automation Computing Continuum*	5.2- Monitorización y observabilidad
	6.4- Explotación de los elementos esenciales del Meta-OS para soportar la innovación en despliegues operativos reales
Scale Model Showcase to Drive Policy Making via IoT Composite Indices and Low-Resource Equipment at the Edge of the Computing Continuum	5.2.2- Gestor de contexto y materialización del modelo de datos
Leveraging IoT and prediction techniques to monitor COVID-19 restrictions in port terminals	5.2.3- Monitorización de los elementos clave
Federation of distributed domains in the Cloud-Edge-IoT Continuum	5.3- Federación de dominios del continuo
aerOS architecture (artículo en finalización)	5.4- Orquestación y despliegue de servicios
A Novel Approach for Calculating Real-Time Composite Indicators Relying on Internet of Things and Industrial Data Spaces	6.4- Explotación de los elementos esenciales del Meta-OS para soportar la innovación en despliegues operativos reales
Autonomous Choreography of WebAssembly Workloads in the Federated Cloud-Edge-IoT Continuum	7.1- Virtualización alternativa y *hardware* de código abierto

Overview of Current Challenges in Multi-Architecture Software Engineering and a Vision for the Future	7.1- Virtualización alternativa y *hardware* de código abierto

Referencias

[1] "Tamaño del mercado de dispositivos IoT y análisis de acciones - Informe de investigación de la industria." , Online: https://www.mordorintelligence.com/es/industry-reports/iot-devices-market.

[2] "Doce tendencias tecnológicas para 2025, el comienzo de la era de la convergencia." , Online: https://elpais.com/tecnologia/2024-12-24/doce-tendencias-tecnologicas-para-2025-el-comienzo-de-la-era-de-la-convergencia.html.

[3] "Mercado de computación en la nube tamaño, crecimiento, informe y análisis." , Online: https://www.mordorintelligence.com/es/industry-reports/cloud-computing-market.

[4] I. Lacalle *et al.*, "Tactile Internet in Internet of Things Ecosystems," in *Lecture Notes in Electrical Engineering*, 2022, vol. 894 LNEE, pp. 794–807.

[5] R. Vaño, I. Lacalle, P. Sowinski, R. S-Julián, and C. E. Palau, "Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions," *Sensors*, vol. 23, no. 4, p. 2215, Feb. 2023.

[6] "La historia de Kubernetes." , Online: https://www.ibm.com/es-es/think/topics/kubernetes-history.

[7] A. Dauda, O. Flauzac, and F. Nolot, "A Survey on IoT Application Architectures," *Sensors 2024, Vol. 24, Page 5320*, vol. 24, no. 16, p. 5320, Aug. 2024.

[8] "Autonomous, scalablE, tRustworthy, intelligent European meta Operating System for the IoT edge-cloud continuum (aerOS) HE Project," 01-Sep-2022. , Online: https://cordis.europa.eu/project/id/101069732.

[9] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on OSI model," in *2011 IEEE 3rd International Conference on Communication Software and Networks, ICCSN 2011*, 2011, pp. 554–557.

[10] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An Overview on Edge Computing Research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020.

[11] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," *GIoTS 2017 - Global Internet of Things Summit, Proceedings*, Aug. 2017.

[12] R. Salama, F. Al-Turjman, M. Aeri, and S. P. Yadav, "Internet of Intelligent Things (IoT) - An Overview," in *2023 International Conference on Computational Intelligence, Communication Technology and Networking (CICTN)*, 2023, pp. 801–805.

[13] A. Al-Dulaimy *et al.*, "The computing continuum: From IoT to the cloud," *Internet of Things*, vol. 27, p. 101272, Oct. 2024.

[14] "AIOTI – Alliance for AI, IoT and Edge Continuum Innovation." , Online: https://aioti.eu/.

[15] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The Seminal Role of Edge-Native Applications," in *2019 IEEE International Conference on Edge Computing, EDGE 2019 - Part of the 2019 IEEE World Congress on Services*, 2019, pp. 33–40.

[16] S. Deng *et al.*, "Cloud-Native Computing: A Survey From the Perspective of Services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024.

[17] P. Raj, S. Vanga, and A. Chaudhary, "Delineating Cloud-Native Edge Computing," in *Cloud-native Computing: How to Design, Develop, and Secure Microservices and Event-Driven Applications*, Wiley-IEEE Press, 2023, pp. 171–201.

[18] "LF Edge projects." , Online: https://www.lfedge.org/projects/.

[19] "The Eclipse Foundation: Edge and IoT." , Online: https://www.eclipse.org/topics/edge-and-iot/.

[20] "CNCF Cloud Native Landscape." , Online: https://landscape.cncf.io/.

[21] "Shaping Europe's digital future: IoT and the Future of Edge Computing in Europe." , Online: https://digital-strategy.ec.europa.eu/en/news/iot-and-future-edge-computing-europe.

[22] "European comission funding & tender opportunities: 'Future European

platforms for the Edge: Meta Operating Systems (RIA).'" , Online: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/horizon-cl4-2021-data-01-05.

[23] "European comission funding & tender opportunities: 'Cognitive Cloud: AI-enabled computing continuum from Cloud to Edge (RIA).'" , Online: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/horizon-cl4-2022-data-01-02.

[24] "EUCloudEdgeIOT: building the European Cloud Edge IoT Continuum for Business and Research." , Online: https://eucloudedgeiot.eu/.

[25] A. Kapadia, B. Wick, J. Roberts, and K. Goldenring, "Edge Native Applications Principles Whitepaper," Jan. 2023, Online: https://www.cncf.io/reports/edge-native-applications-principles-whitepaper/.

[26] F. Brockners, J. Roberts, K. Goldenring, and A. Anderson, "Edge Native Application Design Behaviors Whitepaper," Oct. 2023.

[27] "AWS IoT Greengrass." , Online: https://aws.amazon.com/greengrass/features/.

[28] "What Is AWS Snowball Edge? - AWS Snowball Edge Developer Guide." , Online: https://docs.aws.amazon.com/snowball/latest/developer-guide/whatisedge.html.

[29] "Microsoft Azure IoT Edge." , Online: https://azure.microsoft.com/en-us/products/iot-edge/#iotedge-overview.

[30] "Microsoft Azure Stack Edge." , Online: https://azure.microsoft.com/en-us/products/azure-stack/edge/.

[31] "Google Distributed Cloud." , Online: https://cloud.google.com/distributed-cloud.

[32] I. Lacalle, A. Belsa, R. Vaño, and C. E. Palau, "Framework and Methodology for Establishing Port-City Policies Based on Real-Time Composite Indicators and IoT: A Practical Use-Case," *Sensors*, vol. 20, no. 15, p. 4131, 2020.

[33] "FIWARE Orion Context Broker." , Online: https://fiware-orion.readthedocs.io/en/master/.

[34] "FIWARE-NGSI v2 Specification." , Online: https://fiware-

ges.github.io/orion/archive/api/v2/.

[35] "ETSI Industry Specification Group (ISG) cross cutting Context Information Management (CIM)." , Online: https://www.etsi.org/committee/cim.

[36] "JSON-LD: JSON for Linking Data." , Online: https://json-ld.org/.

[37] "Semantic Web Standards: Resource Description Framework (RDF)." , Online: https://www.w3.org/RDF/.

[38] ETSI, "GS CIM 009 - V1.8.1 - Context Information Management (CIM); NGSI-LD API," Mar. 2024, Online: https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx.

[39] aerOS Project, "D4.2-Software for delivering intelligence at the edge intermediate release," 2024.

[40] "FIWARE Orion-LD." , Online: https://github.com/FIWARE/context.Orion-LD.

[41] "NEC Scorpio NGSI-LD Context Broker." , Online: https://github.com/ScorpioBroker/ScorpioBroker.

[42] "FIWARE Smart Data Models Initiative." , Online: https://www.fiware.org/smart-data-models/.

[43] "Marine Transport Smart Data Model." , Online: https://github.com/smart-data-models/dataModel.MarineTransport.

[44] R. Vaño, I. Lacalle, and C. E. Palau, "Federation of distributed domains in the Cloud-Edge-IoT Continuum," in *2nd International Workshop on MetaOS for the Cloud-Edge-IoT Continuum (MECC '25)*, 2025, pp. 14–19.

[45] I. Lacalle *et al.*, "A Novel Approach to Self-∗ Capabilities in IoT Industrial Automation Computing Continuum," in *2023 IEEE World Forum on Internet of Things (WF-IoT)*, 2023.

[46] R. Busetti, N. El Ioini, H. R. Barzegar, and C. Pahl, "A Comparison of Synchronous and Asynchronous Distributed Particle Swarm Optimization for Edge Computing," in *13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, 2023, pp. 194–203.

[47] "What is gRPC?" , Online: https://grpc.io/docs/what-is-grpc/.

[48] M. Bolanowski *et al.*, "Eficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems," *Frontiers in Artificial Intelligence and Applications*, vol. 355, pp. 97–108, Sep. 2022.

[49] "Protocol Buffers Documentation." , Online: https://protobuf.dev/.

[50] "AsyncAPI: Coming from OpenAPI." , Online: https://www.asyncapi.com/docs/tutorials/getting-started/coming-from-openapi.

[51] "Apache Kafka." , Online: https://kafka.apache.org/.

[52] "Streaming data: Challenges, use cases, and considerations." , Online: https://www.redpanda.com/blog/streaming-data-examples-best-practices-tools.

[53] "KRaft: Apache Kafka Without ZooKeeper." , Online: https://developer.confluent.io/learn/kraft/.

[54] "Redpanda vs Kafka." , Online: https://www.redpanda.com/compare/redpanda-vs-kafka.

[55] "NATS broker Overview." , Online: https://docs.nats.io/nats-concepts/overview.

[56] "NATS Comparison to Kafka, Rabbit, gRPC, and others." , Online: https://docs.nats.io/nats-concepts/overview/compare-nats.

[57] "NATS Adaptive Deployment Architectures." , Online: https://docs.nats.io/nats-concepts/service_infrastructure/adaptive_edge_deployment.

[58] Y. Zhao, W. Wang, Y. Li, C. Colman Meixner, M. Tornatore, and J. Zhang, "Edge Computing and Networking: A Survey on Infrastructures and Applications," *IEEE Access*, vol. 7, pp. 101213–101230, 2019.

[59] X. Kong, Y. Wu, H. Wang, and F. Xia, "Edge Computing for Internet of Everything: A Survey," *IEEE Internet Things J*, vol. 9, no. 23, pp. 23472–23485, Dec. 2022.

[60] "Lightway Core: a modern VPN protocol by ExpressVPN." , Online: https://github.com/expressvpn/lightway-core.

[61] J. A. Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel," in *Network and Distributed System Security (NDSS) Symposium 2017*, 2017.

[62] C. Zanasi, S. Russo, and M. Colajanni, "Flexible zero trust architecture for the cybersecurity of industrial IoT infrastructures," *Ad Hoc Networks*, vol. 156, p. 103414, Apr. 2024.

[63] "Container Network Interface (CNI)." , Online: https://www.cni.dev/.

[64] “Kubernetes Network Plugins.” , Online: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[65] “Cilium documentation: WireGuard Transparent Encryption.” , Online: https://docs.cilium.io/en/latest/security/network/encryption-wireguard/.

[66] “kube-vip documentation.” , Online: https://kube-vip.io/.

[67] P. Srisuresh and K. Egevang, “RFC 3022: Traditional IP Network Address Translator (Traditional NAT),” Jan-2001. , Online: https://www.rfc-editor.org/info/rfc3022.

[68] Tailscale, “How NAT traversal works,” Online: https://tailscale.com/blog/how-nat-traversal-works.

[69] “Tailscale: How it works.” , Online: https://tailscale.com/blog/how-tailscale-works.

[70] “What is libp2p.” , Online: https://docs.libp2p.io/concepts/introduction/overview/.

[71] “Technical specifications for the libp2p networking stack.” , Online: https://github.com/libp2p/specs/.

[72] P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002, vol. 2429, pp. 53–65.

[73] “libp2p documentation: Kademlia DHT.” , Online: https://docs.libp2p.io/concepts/discovery-routing/kaddht/.

[74] “The Distributed Hash Table.” , Online: https://pl-launchpad.io/curriculum/libp2p/dht/.

[75] “libp2p Circuit Relay v2 specification.” , Online: https://github.com/libp2p/specs/blob/master/relay/circuit-v2.md.

[76] M. Seemann, M. Inden, and D. Vyzovitis, “Decentralized Hole Punching,” in *Proceedings of the 2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, vol. 1.

[77] “Components to measure Direct Connection Upgrade through Relay (DCUtR) performance.” , Online: https://github.com/libp2p/punchr.

[78] “libp2p and Ethereum (the Merge).” , Online: https://blog.libp2p.io/libp2p-and-ethereum/.

[79] "EdgeVPN: an immutable, decentralized and statically built p2p VPN." , Online: https://github.com/mudler/edgevpn.

[80] "Kairos P2P Network." , Online: https://kairos.io/docs/architecture/network/.

[81] " Kairos documentation: deploying a High-Availability K3s Cluster with KubeVIP." , Online: https://kairos.io/docs/examples/multi-node-p2p-ha-kubevip/.

[82] J. Alonso *et al.*, "Understanding the challenges and novel architectural models of multi-cloud native applications – a systematic literature review," *Journal of Cloud Computing*, vol. 12, no. 1, pp. 1–34, Dec. 2023.

[83] A. Bhardwaj and C. R. Krishna, "Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey," *Arab J Sci Eng*, vol. 46, no. 9, pp. 8585–8601, Apr. 2021.

[84] "¿Qué son los contenedores? | IBM." , Online: https://www.ibm.com/es-es/topics/containers.

[85] P. J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource Management in a Containerized Cloud: Status and Challenges," *Journal of Network and Systems Management 2019 28:2*, vol. 28, no. 2, pp. 197–246, Nov. 2020.

[86] A. K. Yadav, M. L. Garg, and Ritika, "Docker containers versus virtual machine-based virtualization," *Advances in Intelligent Systems and Computing*, vol. 814, pp. 141–150, 2019.

[87] Canonical Ltd., "Linux Containers: LXC." , Online: https://linuxcontainers.org/lxc/introduction/.

[88] "Docker: Accelerated Container Application Development." , Online: https://www.docker.com/.

[89] "pfSense: World's Most Trusted Open Source Firewall." , Online: https://www.pfsense.org/.

[90] "OPNsense is an open source, feature rich firewall and routing platform, offering cutting-edge network protection." , Online: https://opnsense.org/.

[91] A. Fornés-Leal *et al.*, "Evolution of MANO Towards the Cloud-Native Paradigm for the Edge Computing," in *Advanced Computing and Intelligent Technologies. Lecture Notes in Electrical Engineering*, 2022, vol. 914, pp. 1–16.

[92] “ OpenStack: Open Source Cloud Computing Infrastructure.” , Online: https://www.openstack.org/.

[93] “KubeVirt: building a virtualization API for Kubernetes.” , Online: https://kubevirt.io/.

[94] “ETSI ISG NFV (Network Functions Virtualisation).” , Online: https://www.etsi.org/technologies/nfv.

[95] “ETSI Open Source MANO.” , Online: https://osm.etsi.org/.

[96] “Open Container Initiative.” , Online: https://opencontainers.org/.

[97] “Docker Engine.” , Online: https://docs.docker.com/engine/.

[98] “containerd: an industry-standard container runtime with an emphasis on simplicity, robustness and portability.” , Online: https://containerd.io/.

[99] “runc: CLI tool for spawning and running containers according to the OCI specification.” , Online: https://github.com/opencontainers/runc.

[100] “Moby project.” , Online: https://mobyproject.org/.

[101] “Podman container engine.” , Online: https://podman.io/.

[102] “crun: A fast and lightweight fully featured OCI runtime and C library for running containers.” , Online: https://github.com/containers/crun.

[103] “youki: A container runtime written in Rust.” , Online: https://github.com/containers/youki.

[104] “Container Runtime Interface (CRI).” , Online: https://kubernetes.io/docs/concepts/architecture/cri/.

[105] “CRI-O: Lightweight container runtime for Kubernetes.” , Online: https://cri-o.io/.

[106] “What’s new in CRI-O 1.31.” , Online: https://www.cncf.io/blog/2024/09/12/whats-new-in-cri-o-1-31/.

[107] “iSulad: a light weight container runtime daemon for IOT and Cloud infrastructure.” , Online: https://github.com/openeuler-mirror/iSulad.

[108] “openEuler: OS for Digital Infrastructure.” , Online: https://www.openeuler.org/en/.

[109] “contaiNERD CTL: Docker-compatible CLI for containerd.” , Online: https://github.com/containerd/nerdctl.

[110] “EVE Kubernetes Control Plane Integration - Draft.” , Online:

https://wiki.lfedge.org/display/EVE/EVE+Kubernetes+Control+Plane+Integration+-+Draft.

[111] "balenaOS - Run Docker containers on embedded IoT devices." , Online: https://www.balena.io/os/.

[112] "The Yocto Project." , Online: https://www.yoctoproject.org/.

[113] "Yocto Project." , Online: https://www.yoctoproject.org/.

[114] "balenaEngine: a container engine purpose-built for IoT devices." , Online: https://www.balena.io/engine/.

[115] "balenaOS Compatible Devices." , Online: https://www.balena.io/devices.

[116] "OpenBalena: open source software to manage connected IoT devices at scale." , Online: https://github.com/balena-io/open-balena.

[117] A. Buehrle, "Why Embedded Linux Needs a Container Manager Written in C." , Online: https://pantacor.com/blog/embedded-linux-need-container-manager/.

[118] "Pantavisor: a framework for containerized embedded Linux." , Online: https://pantavisor.io/.

[119] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," *Applied Sciences 2019, Vol. 9, Page 931*, vol. 9, no. 5, p. 931, Mar. 2019.

[120] "Amazon Elastic Kubernetes Service (EKS)." , Online: https://aws.amazon.com/es/eks/.

[121] "Red Hat OpenShift Kubernetes Engine." , Online: https://www.redhat.com/en/technologies/cloud-computing/openshift/kubernetes-engine.

[122] "Certified Kubernetes Conformance Program: Terms and Conditions." , Online: https://github.com/cncf/k8s-conformance/blob/master/terms-conditions/Certified_Kubernetes_Terms.md.

[123] "Kubernetes Cluster Architecture." , Online: https://kubernetes.io/docs/concepts/architecture/.

[124] "Kubernetes command line tool (kubectl)." , Online: https://kubernetes.io/docs/reference/kubectl/.

[125] D. Balouek-Thomert, E. G. Renart, A. R. Zamani, A. Simonet, and M.

Parashar, “Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows,” *International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1159–1174, Nov. 2019.

[126] A. Marchese and O. Tomarchio, “Application and Infrastructure-Aware Orchestration in the Cloud-to-Edge Continuum,” in *2023 IEEE 16th International Conference on Cloud Computing, CLOUD*, 2023, vol. 2023-July, pp. 262–271.

[127] P. Kayal, “Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope,” in *IEEE World Forum on Internet of Things, WF-IoT 2020*, 2020.

[128] A. Jeffery, H. Howard, and R. Mortier, “Rearchitecting Kubernetes for the Edge,” in *4th International Workshop on Edge Systems, Analytics and Networking, EdgeSys 2021, Part of EuroSys 2021*, 2021, pp. 7–12.

[129] “K3s: Lightweight Kubernetes.” , Online: https://k3s.io/.

[130] “Raspberry Pi.” , Online: https://www.raspberrypi.com/.

[131] “Kits y módulos de desarrollador de sistemas integrados de NVIDIA Jetson.” , Online: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/.

[132] “K3OS: the Kubernetes Operating System.” , Online: https://k3os.io/.

[133] “Elemental: Immutable Linux for Rancher.” , Online: https://elemental.docs.rancher.com/.

[134] “MicroK8s - Zero-ops Kubernetes for developers, edge and IoT.” , Online: https://microk8s.io/.

[135] ASSIST-IoT project, “D6.5 Technical Support Documentation - Initial,” Apr. 2022, Online: https://assist-iot.eu/wp-content/uploads/2022/05/D6.5_Technical_Support_Documentation_Initial.pdf.

[136] V. Kjorveziroski and S. Filiposka, “Kubernetes distributions for the edge: serverless performance evaluation,” *Journal of Supercomputing*, vol. 78, no. 11, pp. 13728–13755, Jul. 2022.

[137] M. Usman, S. Ferlin, and A. Brunstrom, “Performance Analysis of Lightweight Container Orchestration Platforms for Edge-Based IoT Applications,” *Proceedings - 2024 IEEE/ACM Symposium on Edge*

Computing, SEC 2024, pp. 321–332, 2024.

[138] "k0s: Kubernetes distribution for bare-metal, on-prem, edge, IoT." , Online: https://k0sproject.io/.

[139] I. Čilić, P. Krivić, I. Podnar Žarko, and M. Kušek, "Performance Evaluation of Container Orchestration Tools in Edge Computing Environments," *Sensors 2023, Vol. 23, Page 4008*, vol. 23, no. 8, p. 4008, Apr. 2023.

[140] L. M. Saini, P. Ajmani, V. Asha, K. Pithamber, and N. Sreevani, "KubeEdge for Scalable IoT Applications," in *2024 7th International Conference on Contemporary Computing and Informatics (IC3I)*, 2024, pp. 1–7.

[141] "KubeEdge: a Kubernetes Native Edge Computing Framework." , Online: https://kubeedge.io/en/.

[142] "KubeEdge EdgeMesh: Simplified network and services for edge applications." , Online: https://github.com/kubeedge/edgemesh.

[143] W. Xu, "Test Report on KubeEdge's Support for 100,000 Edge Nodes | KubeEdge." , Online: https://kubeedge.io/en/blog/scalability-test-report/.

[144] "Kubernetes on Edge Day 2021: KubeEdge and Kubernetes help manage alll the monitoring devices on the world's longest cross-sea bridge." , Online: https://kubenetesedgedayeu21.sched.com/event/iS2I/kubeedge-and-kubernetes-help-manage-all-the-monitoring-devices-on-the-worlds-longest-cross-sea-bridge-huan-wei-harmonycloud.

[145] "Sedna documentation." , Online: https://sedna.readthedocs.io/en/latest/.

[146] "OpenYurt: An open platform that extends upstream Kubernetes to Edge." , Online: https://openyurt.io/.

[147] "SuperEdge: an open-source container management system for edge computing. It extends native Kubernetes to the edge in a non-intrusive wAY." , Online: https://superedge.io/.

[148] "EdgeX Foundry: the Enabled Open Software Platform." , Online: https://www.edgexfoundry.org/.

[149] "Open Horizon - LF Edge." , Online: https://www.lfedge.org/projects/openhorizon/.

[150] "Baetyl: Baetyl, extend cloud computing, data and service seamlessly to edge devices." , Online: https://baetyl.io/en/.

[151] "Eclipse ioFog." , Online: https://iofog.org/.

[152] “Akri: a Kubernetes Resource Interface for the Edge.” , Online: https://docs.akri.sh/.

[153] “Akri: Custom Discovery Handlers.” , Online: https://github.com/project-akri/akri-docs/blob/main/docs/development/handler-development.md.

[154] “OpenFaaS: Serverless Functions Made Simple.” , Online: https://www.openfaas.com/.

[155] “Knative: an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications.” , Online: https://knative.dev/docs/.

[156] “CloudEvents: a specification for describing event data in a common way.” , Online: https://cloudevents.io/.

[157] “buildah: A tool that facilitates building OCI images.” , Online: https://github.com/containers/buildah.

[158] “Cloud Native Buildpacks.” , Online: https://buildpacks.io/.

[159] “Docker Hub Container Image Library.” , Online: https://hub.docker.com/.

[160] “Harbor open source registry.” , Online: https://goharbor.io/.

[161] “Dragonfly.” , Online: https://d7y.io/.

[162] “Docker Compose.” , Online: https://docs.docker.com/compose/.

[163] “Kustomize: Kubernetes native configuration management.” , Online: https://kustomize.io/.

[164] “Helm: the package manager for Kubernetes.” , Online: https://helm.sh/.

[165] “Artifact Hub: Cloud Native packages repository.” , Online: https://artifacthub.io/.

[166] “Kubernetes Operator pattern.” , Online: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[167] S. Sultan, I. Ahmad, and T. Dimitriou, “Container security: Issues, challenges, and the road ahead,” *IEEE Access*, vol. 7, pp. 52976–52996, 2019.

[168] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, “Consolidate IoT Edge Computing with Lightweight Virtualization,” *IEEE Netw*, vol. 32, no. 1, pp. 102–111, Jan. 2018.

[169] R. Kumar and B. Thangaraju, “Performance Analysis between RunC and Kata Container Runtime,” in *6th IEEE International Conference on Electronics, Computing and Communication Technologies, CONECCT 2020*, 2020.

[170] "Kata Containers: open-source container runtime, building lightweight virtual machines that seamlessly plug into the containers ecosystem ." , Online: https://katacontainers.io/.

[171] Z. Li *et al.*, "RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing," 2022.

[172] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Queue*, vol. 11, no. 11, Dec. 2013.

[173] I. Sarrigiannis, L. M. Contreras, K. Ramantas, A. Antonopoulos, and C. Verikoukis, "Fog-Enabled Scalable C-V2X Architecture for Distributed 5G and beyond Applications," *IEEE Netw*, vol. 34, no. 5, pp. 120–126, Sep. 2020.

[174] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," *Proceedings - 8th IEEE International Symposium on Cloud and Services Computing, SC2 2018*, pp. 1–8, Dec. 2018.

[175] "MirageOS: a programming framework for building type-safe, modular systems." , Online: https://mirage.io/.

[176] "Nabla containers: a new approach to container isolation." , Online: https://nabla-containers.github.io/.

[177] "runnc: OCI-interfacing Container runtime for Nabla Containers." , Online: https://github.com/nabla-containers/runnc.

[178] "UNICORE EU H2020 project." , Online: https://unicore-project.eu/.

[179] S. Kuenzer *et al.*, "Unikraft: Fast, Specialized Unikernels the Easy Way," in *Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.

[180] "Unikraft Kraftkit: build and use highly customized and ultra-lightweight unikernel VMs." , Online: https://github.com/unikraft/kraftkit.

[181] S. F. Ion, C. Carabas, N. Tapus, and D. Ciorba, "Enhancing Blockchain Performance via Unikraft: A Case Study of Implementation and Scalability Analysis on MultiversX," in *23rd RoEduNet IEEE International Conference*, 2024.

[182] "Solomon Hykes publication in Twitter about WASM and WASI." , Online: https://x.com/solomonstre/status/1111004913222324225?lang=en.

[183] "WebAssembly." , Online: https://webassembly.org/.

[184] "WebAssembly Specification Release 2.0 (Draft 2025-01-28)." , Online:

https://webassembly.github.io/spec/core/.

[185] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *2019 USENIX Annual Technical Conference*, 2019, pp. 107–120.

[186] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, vol. 11543 LNCS, pp. 23–42.

[187] "WebAssembly: The Next Frontier in Cloud-Native Evolution." , Online: https://wasmcloud.com/blog/webassembly-the-next-frontier-in-cloud-native-evolution/.

[188] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "WebAssembly as a Common Layer for the Cloud-edge Continuum," in *2nd Workshop on Flexible Resource and Application Management on the Edge, FRAME 2022, co-located with HPDC 2022*, 2022, pp. 3–8.

[189] N. Mäkitalo *et al.*, "WebAssembly Modules as Lightweight Containers for Liquid IoT Applications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2021, vol. 12706 LNCS, pp. 328–336.

[190] "WASI: WebAssembly System Interface." , Online: https://github.com/WebAssembly/WASI.

[191] L. Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web." , Online: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[192] E. Wen and G. Weber, "Wasmachine: Bring the edge up to speed with a webassembly OS," in *IEEE International Conference on Cloud Computing, CLOUD*, 2020, vol. 2020-October, pp. 353–360.

[193] "WASI.dev." , Online: https://wasi.dev/.

[194] "WASI standarization proposals." , Online: https://github.com/WebAssembly/WASI/blob/main/Proposals.md.

[195] "Semantic Versioning 2.0.0." , Online: https://semver.org/.

[196] "Bytecode Alliance: WASI 0.2 Launched." , Online: https://bytecodealliance.org/articles/WASI-0.2.

[197] “The WIT (Wasm Interface Type) language.” , Online: https://component-model.bytecodealliance.org/design/wit.html.

[198] “Specification of the Wasm Interface Type format.” , Online: https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md.

[199] “tinygo: Go compiler for small places.” , Online: https://github.com/tinygo-org/tinygo.

[200] “wit-bindgen: a language binding generator for WebAssembly interface types.” , Online: https://github.com/bytecodealliance/wit-bindgen.

[201] “Bytecode Alliance: announcing WASI 0.2.1.” , Online: https://bytecodealliance.org/articles/WASI-0.2.1.

[202] “CNCF TAG Runtime: Wasm OCI Artifact layout.” , Online: https://tag-runtime.cncf.io/wgs/wasm/deliverables/wasm-oci-artifact/.

[203] “wasm-pkg-tools: tools to package up Wasm components.” , Online: https://github.com/bytecodealliance/wasm-pkg-tools.

[204] “WASI Roadmap.” , Online: https://wasi.dev/roadmap.

[205] “warg: a secure registry protocol for Wasm packages.” , Online: https://warg.io/.

[206] “Wasmer: the Universal WebAssembly Runtime.” , Online: https://wasmer.io/.

[207] “Wasmtime: a fast and secure runtime for WebAssembly.” , Online: https://wasmtime.dev/.

[208] “WasmEdge: bring the cloud-native and serverless application paradigms to Edge Computing.” , Online: https://wasmedge.org/.

[209] “Oracle GraalVM.” , Online: https://www.graalvm.org/.

[210] “GraalWasm.” , Online: https://github.com/oracle/graal/tree/master/wasm.

[211] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni, “Twine: An Embedded Trusted Runtime for WebAssembly,” in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, vol. 2021-April, pp. 205–216.

[212] “Fermyon Spin: develop serverless WebAssembly apps.” , Online: https://www.fermyon.com/spin.

[213] V. Kjorveziroski and S. Filiposka, “WebAssembly Orchestration in the

Context of Serverless Computing," *Journal of Network and Systems Management*, vol. 31, no. 3, pp. 1–24, Jul. 2023.

[214] "wasmCloud: Wasm-native orchestration." , Online: https://wasmcloud.com/.

[215] "Open Application Model Specification: an open model for defining cloud native apps." , Online: https://oam.dev/.

[216] "wasmCloud Lattice." , Online: https://wasmcloud.com/docs/concepts/lattice/.

[217] B. C. Gain, "When WebAssembly Replaces Docker," *The New Stack*, 2022. , Online: https://thenewstack.io/when-webassembly-replaces-docker/.

[218] M. Irwin, "Introducing the Docker+Wasm Technical Preview." , Online: https://www.docker.com/blog/docker-wasm-technical-preview/.

[219] "Announcing Docker+Wasm Technical Preview 2." , Online: https://www.docker.com/blog/announcing-dockerwasm-technical-preview-2/.

[220] "runwasi: a project to facilitate running wasm workloads managed by containerd ." , Online: https://github.com/containerd/runwasi.

[221] "Kubernetes reference documentation: Runtime Class." , Online: https://kubernetes.io/docs/concepts/containers/runtime-class/.

[222] "runtime-class-manager: a Kubernetes operator to manage Runtime Classes." , Online: https://github.com/spinframework/runtime-class-manager.

[223] "Krustlet: run WebAssembly workloads in your Kubernetes cluster." , Online: https://krustlet.dev/.

[224] M. Sebrechts, T. Ramlot, S. Borny, T. Goethals, B. Volckaert, and F. De Turck, "Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI," in *11th IEEE International Conference on Cloud Networking*, 2022.

[225] P. Sowinski, I. Lacalle, R. Vaño, and C. E. Palau, "Autonomous Choreography of WebAssembly Workloads in the Federated Cloud-Edge-IoT Continuum," in *2023 IEEE 12th International Conference on Cloud Networking, CloudNet 2023*, 2023, pp. 454–459.

[226] P. Sowinski, I. Lacalle, R. Vaño, C. E. Palau, M. Ganzha, and M. Paprzycki, "Overview of Current Challenges in Multi-architecture Software Engineering and a Vision for the Future," in *Big Data Analytics in Astronomy, Science,*

and Engineering (BDA 2024), 2025, vol. 15546, pp. 74–94.

[227] "universAAL IoT platform." , Online: https://www.universaal.info/.

[228] "Sofia2 1.0 documentation." , Online: https://sofia2.readthedocs.io/en/latest/.

[229] A. Belsa Pellicer, "Especificación y desarrollo de mecanismos de interoperabilidad a nivel de Middleware y Aplicaciones/Servicios entre Plataformas Heterogéneas de Internet de las Cosas," Universitat Politècnica de València, Valencia (Spain), 2022.

[230] "Interoperability of Heterogeneous IoT Platforms (INTER-IoT) H2020 Project," 01-Jan-2016. , Online: https://cordis.europa.eu/project/id/687283/de.

[231] "ACTivating InnoVative IoT smart living environments for AGEing well (ACTIVAGE) H2020 Project," 01-Jan-2017. , Online: https://cordis.europa.eu/project/id/732679/es.

[232] "Port IoT for Environmental Leverage (PIXEL) H2020 Project," 01-May-2018. , Online: https://cordis.europa.eu/project/id/769355.

[233] "How tomorrow's ports can boost economies, create jobs and protect nature | World Economic Forum." , Online: https://www.weforum.org/stories/2025/02/how-ports-can-boost-economies-create-jobs-and-protect-nature/.

[234] C. E. Palau *et al.*, "D6.2 PIXEL Information System architecture and design - Version 2," 2019. , Online: https://pixel-ports.eu/wp-content/uploads/2020/05/D6.2-PIXEL-Information-System-architecture-and-design-v2.pdf.

[235] "Elasticsearch: un motor de búsqueda y analítica de RESTful distribuido." , Online: https://www.elastic.co/es/elasticsearch.

[236] "pyngsi: NGSI Python framework intended to build a Fiware NGSI Agent." , Online: https://pypi.org/project/pyngsi/.

[237] J. A. Pérez *et al.*, "D6.4 PIXEL data acquisition, information hub and data representation v2," Jun. 2019, Online: https://pixel-ports.eu/wp-content/uploads/2020/07/D6.4_PIXEL-data-acquisition-information-hub-and-data-representation-v2.pdf.

[238] T. Milošević *et al.*, "The Port Environmental Index: A Quantitative IoT-Based Tool for Assessing the Environmental Performance of Ports," *Journal*

of Marine Science and Engineering 2023, Vol. 11, Page 1969, vol. 11, no. 10, p. 1969, Oct. 2023.

[239] "FIWARE Smart Data Models: KeyPerformanceIndicator." , Online: https://github.com/smart-data-models/dataModel.KeyPerformanceIndicator/blob/master/KeyPerformanceIndicator/README.md.

[240] I. Lacalle Úbeda, "Diseño y desarrollo de una arquitectura de Internet de las Cosas de Nueva Generación orientada al cálculo y predicción de índices compuestos aplicada en entornos reales," Universitat Politècnica de València, Valencia (Spain), 2022.

[241] "Next Generation IoT." , Online: https://ngiot.eu/.

[242] "Architecture for Scalable, Self-*, human-centric, Intelligent, Secure, and Tactile next generation IoT (ASSIST-IoT) H2020 Project," 01-Nov-2020. , Online: https://cordis.europa.eu/project/id/957258.

[243] A. Fornés-Leal *et al.*, "ASSIST-IoT: A Reference Architecture for Next Generation Internet of Things," in *21st International Conference on Intelligent Software Methodologies, Tools, and Techniques, SOMET 2022*, 2022, vol. 355, pp. 109–128.

[244] ASSIST-IoT project, "ASSIST-IoT D6.7 Release and Distribution Plan-Final," 2023.

[245] "EU-funded Innovations: Tactile multi-access network IoT gateway and edge node to support AI and analytics (GWEN)." , Online: https://innovation-radar.ec.europa.eu/innovation/58299.

[246] F. Mahedero Biot *et al.*, "A Novel Orchestrator Architecture for Deploying Virtualized Services in Next-Generation IoT Computing Ecosystems," *Sensors 2025, Vol. 25, Page 718*, vol. 25, no. 3, p. 718, Jan. 2025.

[247] Open Continuum, "D4.3 TOWARDS A EUROPEAN ECOSYSTEM FOR THE COMPUTING CONTINUUM," 2023.

[248] "ISO/IEC JTC 1/SC 41 - Internet of things and digital twin." , Online: https://www.iso.org/committee/6483279.html.

[249] Open Continuum / EUCloudEdgeIoT Task Force 3, "Functional View of the Continuum Reference Architecture: Minimum set of expected functionalities," Jun. 2024, Online: https://zenodo.org/records/11656674.

[250] aerOS Project, "D2.7 – aerOS architecture definition (2)," May 2024, Online:

https://aeros-project.eu/wp-content/uploads/2024/07/D2.7.pdf.

[251] "EU Funding & Tenders Portal: Future European platforms for the Edge: Meta Operating Systems (RIA) - HORIZON-CL4-2021-DATA-01-05." , Online: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/horizon-cl4-2021-data-01-05.

[252] "Flexible, scaLable and secUre decentralIzeD Operating System (FLUIDOS) HE Project," 01-Sep-2022. , Online: https://cordis.europa.eu/project/id/101070473.

[253] "Liqo: enable dynamic Kubernetes multi cluster topologies." , Online: https://liqo.io/.

[254] FLUIDOS Project, "D2.1 SCENARIOS, REQUIREMENTS AND REFERENCE ARCHITECTURE," Online: https://www.fluidos.eu/wp-content/uploads/sites/86/2024/06/FLUIDOS_NL2-October-2023.pdf.

[255] S. Galantino *et al.*, "Building the Cloud Continuum with REAR," in *2024 IEEE 10th International Conference on Network Softwarization, NetSoft 2024*, 2024, pp. 67–72.

[256] "Towards a functional continuum operating system (ICOS) HE Project," 01-Sep-2022. , Online: https://cordis.europa.eu/project/id/101070177.

[257] ICOS Project, "D2.4 ICOS Architectural Design," Online: https://www.icos-project.eu/files/deliverables/D2.4-architectural-design-it2.pdf.

[258] ICOS Project, "D4.1 Data management, Intelligence and Security Layers," Online: https://www.icos-project.eu/files/deliverables/D4.1_management_Intelligence_Security_v1.0.pdf.

[259] "Secure Computing Continuum (IoT, Edge, Cloud, Dataspaces)." , Online: https://cordis.europa.eu/programme/id/HORIZON_HORIZON-CL3-2023-CS-01-01.

[260] "Cognitive Computing Continuum: Intelligence and automation for more efficient data processing (AI, data and robotics partnership) (RIA)." , Online: https://cordis.europa.eu/programme/id/HORIZON_HORIZON-CL4-2023-DATA-01-04.

[261] R. S-Julián, I. Lacalle, R. Vaño, F. Boronat, and C. E. Palau, "Self-* Capabilities of Cloud-Edge Nodes: A Research Review," *Sensors 2023, Vol. 23, Page 2931*, vol. 23, no. 6, p. 2931, Mar. 2023.

[262] T. Stober and U. Hansmann, *Best Practices for Large Software Development Projects*. Heidelberg: Springer, 2010.

[263] J. Robertson and S. Robertson, "Volere Requirements Specification Template (Edition 6.1)," 2000.

[264] "GitLab." , Online: https://about.gitlab.com/.

[265] aerOS Project, "D2.5 - DevPrivSecOps methodology specification (2)," Online: https://aeros-project.eu/wp-content/uploads/2024/07/D2.5.pdf.

[266] "¿Qué es GitOps?" , Online: https://about.gitlab.com/es/topics/gitops/.

[267] "awesome-compose: awesome Docker Compose samples." , Online: https://github.com/docker/awesome-compose.

[268] J. J. Garcia *et al.*, "Towards More Explainable and Traceable AI: Gray-boxed Design in a Case of Microservice Allocation," in *18th International Conference on INnovations in Intelligent SysTems and Applications, INISTA 2024*, 2024.

[269] "OASIS TOSCA Version 2.0." , Online: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

[270] "FOAF Vocabulary Specification." , Online: http://xmlns.com/foaf/spec/.

[271] "Learn More about the Context Broker as a CEF Building Block - FIWARE." , Online: https://www.fiware.org/2018/10/22/opportunities-to-learn-more-about-the-context-broker-as-a-cef-building-block/.

[272] "FIWARE Experts Certification." , Online: https://www.fiware.org/community/fiware-experts/.

[273] R. Vaño, I. Lacalle, and C. E. Palau, "Scale Model Showcase to Drive Policy Making via IoT Composite Indices and Low-Resource Equipment at the Edge of the Computing Continuum," in *Global Challenges for a Sustainable Society. EURECA-PRO 2022.Springer Proceedings in Earth and Environmental Sciences*, 2023, vol. Part F639, pp. 447–456.

[274] "MongoDB: The World's Leading Modern Database." , Online: https://www.mongodb.com/.

[275] "FIWARE IoT Agents." , Online: https://github.com/telefonicaid/iotagent-node-lib?tab=readme-ov-file#iot-agents-available.

[276] R. Vaño, I. Lacalle, B. Molina, and C. E. Palau, "Leveraging IoT and prediction techniques to monitor COVID-19 restrictions in port terminals," in *7th IEEE World Forum on Internet of Things, WF-IoT 2021*, 2021, pp.

205–210.

[277] “Prometheus - Monitoring system & time series database.” , Online: https://prometheus.io/.

[278] “Prometheus Node exporter: exporter for machine metrics.” , Online: https://github.com/prometheus/node_exporter.

[279] “cAdvisor: analyzes resource usage and performance characteristics of running containers.” , Online: https://github.com/google/cadvisor.

[280] “Exploring Kepler’s potentials: unveiling cloud application power consumption | CNCF.” , Online: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/.

[281] “TimescaleDB: PostgreSQL for time series and events.” , Online: https://www.timescale.com/.

[282] “FIWARE Mintaka: implementation of the NGSI-LD temporal retrieval api.” , Online: https://github.com/FIWARE/mintaka.

[283] “libp2p Stream Multiplexing.” , Online: https://docs.libp2p.io/concepts/multiplex/overview/.

[284] D. Choi, K. S. Chung, and J. Shon, “An Improvement on the Weighted Least-Connection Scheduling Algorithm for Load Balancing in Web Cluster Systems,” in *Communications in Computer and Information Science*, 2010, vol. 121 CCIS, pp. 127–134.

[285] “linuxserver/docker-wireguard.” , Online: https://github.com/linuxserver/docker-wireguard.

[286] “Dnsmasq - network services for small networks.” , Online: https://thekelleys.org.uk/dnsmasq/doc.html.

[287] Cloud Native Computing Foundation, “CNCF Operator White Paper - Final Version,” 2023, Online: https://github.com/cncf/tag-app-delivery/blob/main/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md.

[288] “NATS Protocol Binding for CloudEvents.” , Online: https://github.com/cloudevents/spec/blob/main/cloudevents/bindings/nats-protocol-binding.md.

[289] “Operator Framework.” , Online: https://operatorframework.io/.

[290] “OperatorHub.io | The registry for Kubernetes Operators.” , Online:

https://operatorhub.io/.

[291] “Sidecar Containers | Kubernetes.” , Online: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/.

[292] “NATS JetStream.” , Online: https://docs.nats.io/nats-concepts/jetstream.

[293] “Keycloak: Open Source Identity and Access Management.” , Online: https://www.keycloak.org/.

[294] “RFC 7636: Proof Key for Code Exchange.” , Online: https://oauth.net/2/pkce/.

[295] “Spring Framework.” , Online: https://spring.io/.

[296] “Socket.IO: bidirectional and low-latency communication for every platform.” , Online: https://socket.io/.

[297] “FIWARE Smart Data Models: Alert Data Model.” , Online: https://github.com/smart-data-models/dataModel.Alert.

[298] “v-network-graph: An interactive network graph visualization component for Vue 3.” , Online: https://dash14.github.io/v-network-graph/.

[299] “Kubernetes Recommended Labels.” , Online: https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/.

[300] “Helm: The Chart Best Practices Guide.” , Online: https://helm.sh/docs/chart_best_practices/.

[301] “Helm chart generator.” , Online: https://github.com/ravaga/helm-chart-generator.

[302] “Bitnami Helm Charts Template.” , Online: https://github.com/bitnami/charts/tree/main/template.

[303] “IOTA: a decentralized blockchain infrastructure.” , Online: https://www.iota.org/.

[304] S. Cuñat Negueroles *et al.*, “A Blockchain-based Digital Twin for IoT deployments in logistics and transportation,” *Future Generation Computer Systems*, vol. 158, pp. 73–88, Sep. 2024.

[305] J. Miguel Ibañez de Aldecoa Quintana, “NIVELES DE MADUREZ DE LA TECNOLOGÍA. TECHNOLOGY READINESS LEVELS. TRLS.”

[306] “SATRD group Infrastructure.” , Online: https://satrd.es/infrastructure/.

[307] “Talos Linux: Linux designed for Kubernetes.” , Online:

https://www.talos.dev/.

[308] "StarFive VisionFive 2 board." , Online: https://www.starfivetech.com/en/site/boards.

[309] "Home Assistant." , Online: https://www.home-assistant.io/.

[310] "Home Assistant Automation: Webhook trigger." , Online: https://www.home-assistant.io/docs/automation/trigger/#webhook-trigger.

[311] "CloudFerro - Cloud computing services." , Online: https://cloudferro.com/#cloud.

[312] "Linked Open Terms (LOT) Methodology." , Online: https://lot.linkeddata.es/.

[313] "aerOS continuum ontology." , Online: https://wp4.pages.aeros-project.eu/t4.1/aeros-continuum/.

[314] "Open5GS: Open Source implementation for 5G Core and EPC." , Online: https://open5gs.org/.

[315] "AMARI Callbox Classic." , Online: https://www.amarisoft.com/test-and-measurement/device-testing/device-products/amari-callbox-classic.

[316] "FIWARE Smart Data Models: Vehicle Data Model." , Online: https://github.com/smart-data-models/dataModel.Transportation/tree/master/Vehicle.

[317] "aerOS Management Portal/MVP Demonstrator." , Online: https://www.youtube.com/watch?v=UV4mnN4CrwI.

[318] "FIWARE Orion-LD functional tests suite." , Online: https://github.com/FIWARE/context.Orion-LD/tree/develop/test/functionalTest.

[319] "FIWARE Orion-LD load tests." , Online: https://github.com/FIWARE/load-tests/tree/master.

[320] "aerOS Pilot 5 Infrastructure & IoT application for Smart, Energy Efficient & Health Safety Buildings ." , Online: https://www.youtube.com/watch?v=lB8GPRL_bRQ&t=1s.

[321] "InfluxDB Time Series Data Platform." , Online: https://www.influxdata.com/.

[322] "aerOS Pilot 5 - Automation and Service Resilience Component." , Online: https://www.youtube.com/watch?v=lTU-H_bxhFM.

[323] "aerOS Pilot 4 at EUROGATE Container Terminal Limassol, Cyprus." , Online: https://www.youtube.com/watch?v=fYC8bgD70MQ.

[324] "Cyprus University of Technology." , Online: https://www.cut.ac.cy/?languageId=1.

[325] "aerOS Pilot 4 - Computer Vision on the Edge." , Online: https://www.youtube.com/watch?v=D_qjqh5CQFo.

[326] "aerOS Pilot 1.1 - Data-Driven Cognitive Production Lines." , Online: https://www.youtube.com/watch?v=7Gm6X1UKbr8.

[327] "EU's Digital Product Passport: Advancing transparency and sustainability." , Online: https://data.europa.eu/en/news-events/news/eus-digital-product-passport-advancing-transparency-and-sustainability.

[328] "aerOS Pilot 1.1 - Green manufacturing and C02 footprint monitoring." , Online: https://www.youtube.com/watch?v=dAZ_OImnSaA.

[329] A. Belsa, R. Vaño, I. Lacalle, M. Julián, F. Boronat, and C. E. Palau, "A Novel Approach for Calculating Real-Time Composite Indicators Relying on Internet of Things and Industrial Data Spaces," in *Intelligent Distributed Computing XIV. IDC 2021. Studies in Computational Intelligence, vol 1026*, 2022, vol. 1026, pp. 45–55.

[330] U. Ahle, J. J. Hierro, U. Ahle, and J. J. Hierro, "FIWARE for Data Spaces," in *Designing Data Spaces*, Springer, Cham, 2022, pp. 395–417.

[331] J. Hierro and S. Steinbuss, "Data Spaces Business Alliance: Unleashing the Data Economy," 2023, Online: https://data-spaces-business-alliance.eu/wp-content/uploads/dlm_uploads/Data-Spaces-Business-Alliance-Technical-Convergence-V2.pdf.

[332] F. Lumpp, F. Barchi, A. Acquaviva, and N. Bombieri, "On the Containerization and Orchestration of RISC-V architectures for Edge-Cloud computing," in *ESAAM 2023: 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*, 2023, pp. 21–28.

[333] "Recommendations and roadmap for European sovereignty on open source hardware, software and RISC-V Technologies." , Online: https://digital-strategy.ec.europa.eu/en/library/recommendations-and-roadmap-european-sovereignty-open-source-hardware-software-and-risc-v.

[334] "European Processor Initiative." , Online: https://www.european-processor-initiative.eu/.

[335] “DARE European Project.” , Online: https://dare-riscv.eu/.

[336] “Exclusive: China to publish policy to boost RISC-V chip use nationwide, sources say | Reuters.” , Online: https://www.reuters.com/technology/china-publish-policy-boost-risc-v-chip-use-nationwide-sources-2025-03-04/?utm_source=reddit.com.

[337] “A Smart and Adaptive Framework for Enhancing Trust in 6G Networks (SAFE 6G) HE Project,” 01-Jan-2024. , Online: https://cordis.europa.eu/project/id/101139031.

[338] “Open CloudEdgeIoT Platform Uptake in Large Scale Cross-Domain Pilots (O-CEI) HE Project,” 01-Jan-2025. , Online: https://cordis.europa.eu/project/id/101189589.

[339] “OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability.” , Online: https://opentelemetry.io/.

[340] “Issues authored by ravaga in FIWARE/context.Orion-LD.” , Online: https://github.com/FIWARE/context.Orion-LD/issues?q=is%3Aissue%20author%3Aravaga.

Apéndices

Apéndice A – Modelo de datos

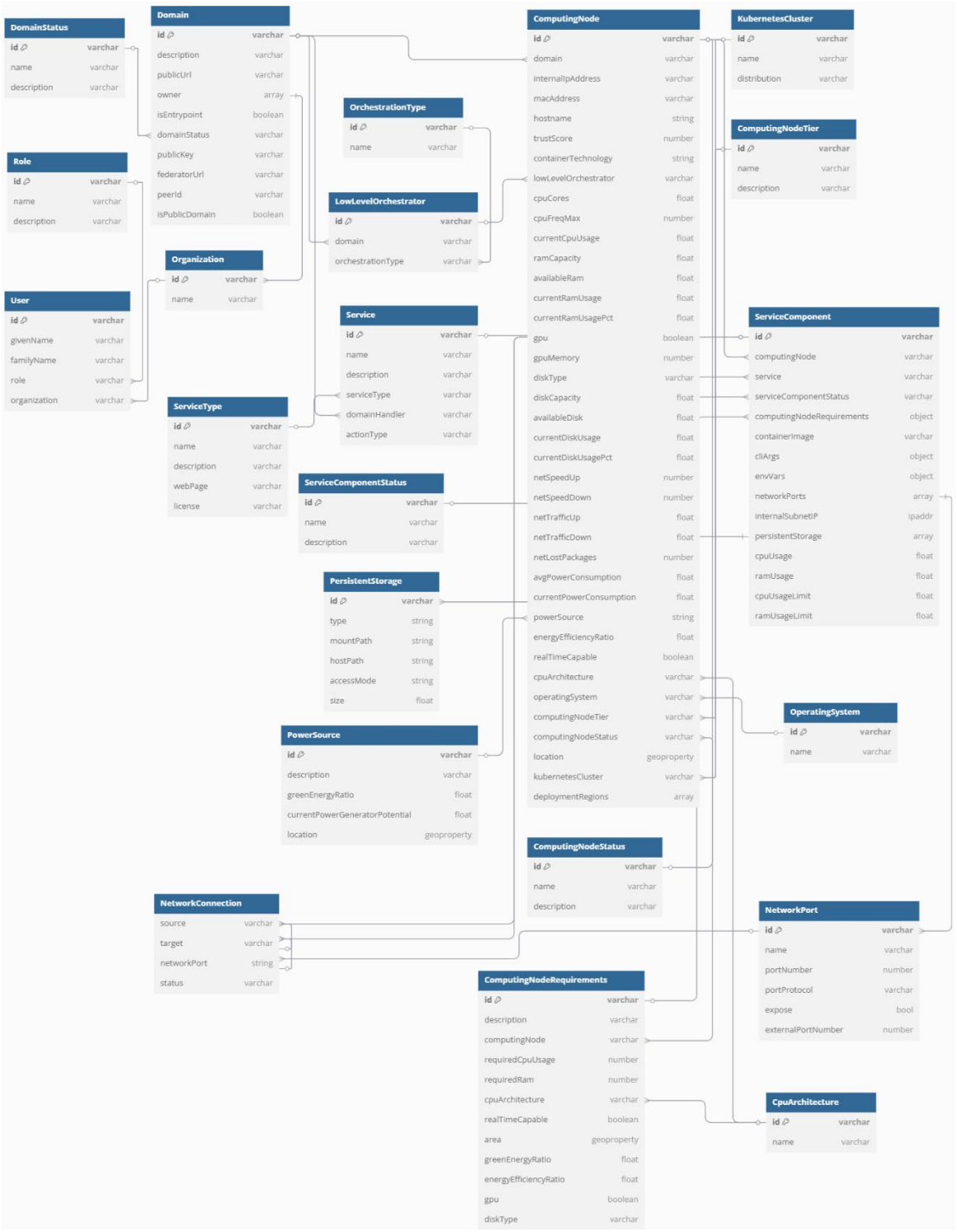


Figura A.1: Modelo de datos completo del continuo de computación CEI

Apéndice B – Diagramas de los componentes desarrollados

Durante el proceso de desarrollo de los componentes pertenecientes a los bloques esenciales que habilitan la creación de un Meta-OS, el doctorando consideró la conveniencia de incluir algunos diagramas ayudaran a representar el *software* desarrollado, pero sin incurrir en los clásicos diagramas UML [1](Unified Modeling Language). Tras analizar las herramientas disponibles en el mercado, se identificó la herramienta Codeviz[2], un *plugin* para el IDE VSCode[3]. Esta herramienta genera automáticamente diagramas interactivos del código fuente mediante la aplicación de técnicas de inteligencia artificial, los cuales permiten visualizar tanto la estructura interna del proyecto como su interacción con otros componentes o bibliotecas externas.

Los diagramas que se muestran en este apéndice fueron inicialmente generados con Codeviz y posteriormente exportados en formato estático como diagramas de la herramienta Draw.io[4], con el objetivo de realizar las modificaciones necesarias de forma manual e interactiva, y así obtener las versiones definitivas que se muestran a continuación.

Por último, con el fin de no sobrecargar esta ya de por sí extensa memoria con contenido redundante, se ha decidido incluir una selección representativa de los componentes desarrollados, concretamente el Federador (ver 5.3.1.1), el Operador del LLO de Kubernetes (revisar 5.4.4.1) y el controlador del LLO de Docker (ver 5.4.4.2).

[1] https://www.uml.org/

[2] https://marketplace.visualstudio.com/items?itemName=CodeViz.codeviz

[3] https://code.visualstudio.com/

[4] https://www.drawio.com/

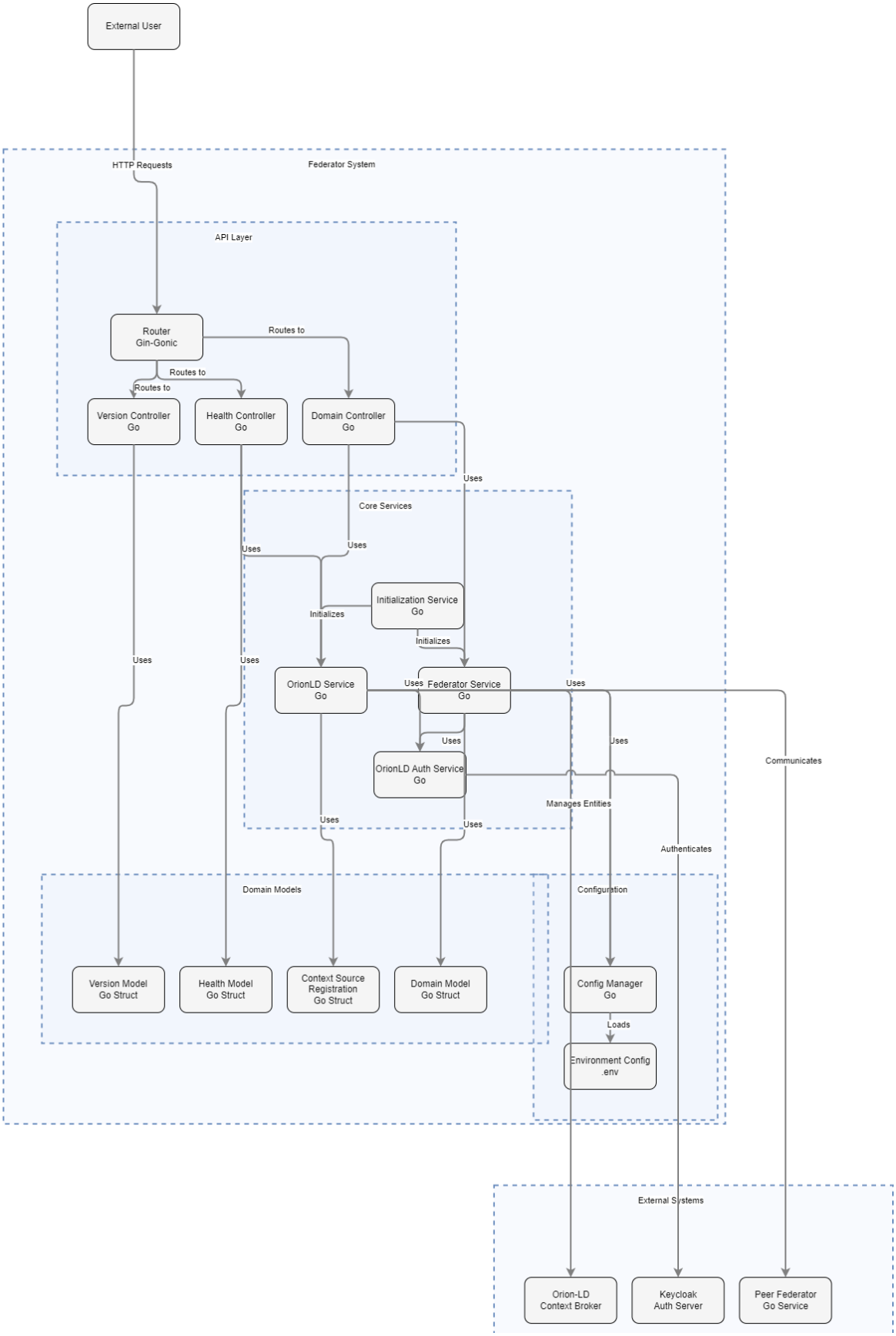


Figura A.2: Diagrama del elemento Federador

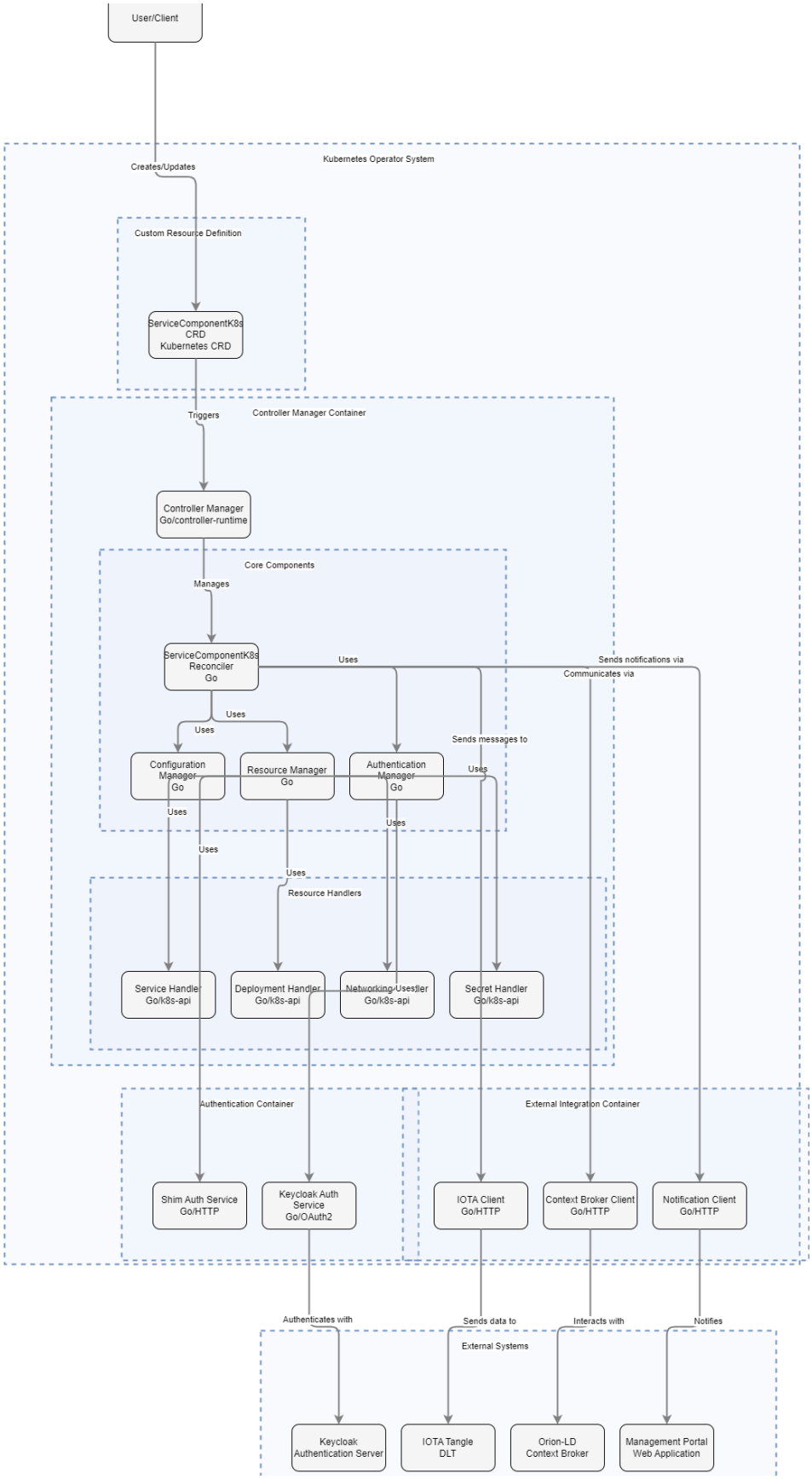


Figura A.3: Diagrama del Operador del LLO de K8s

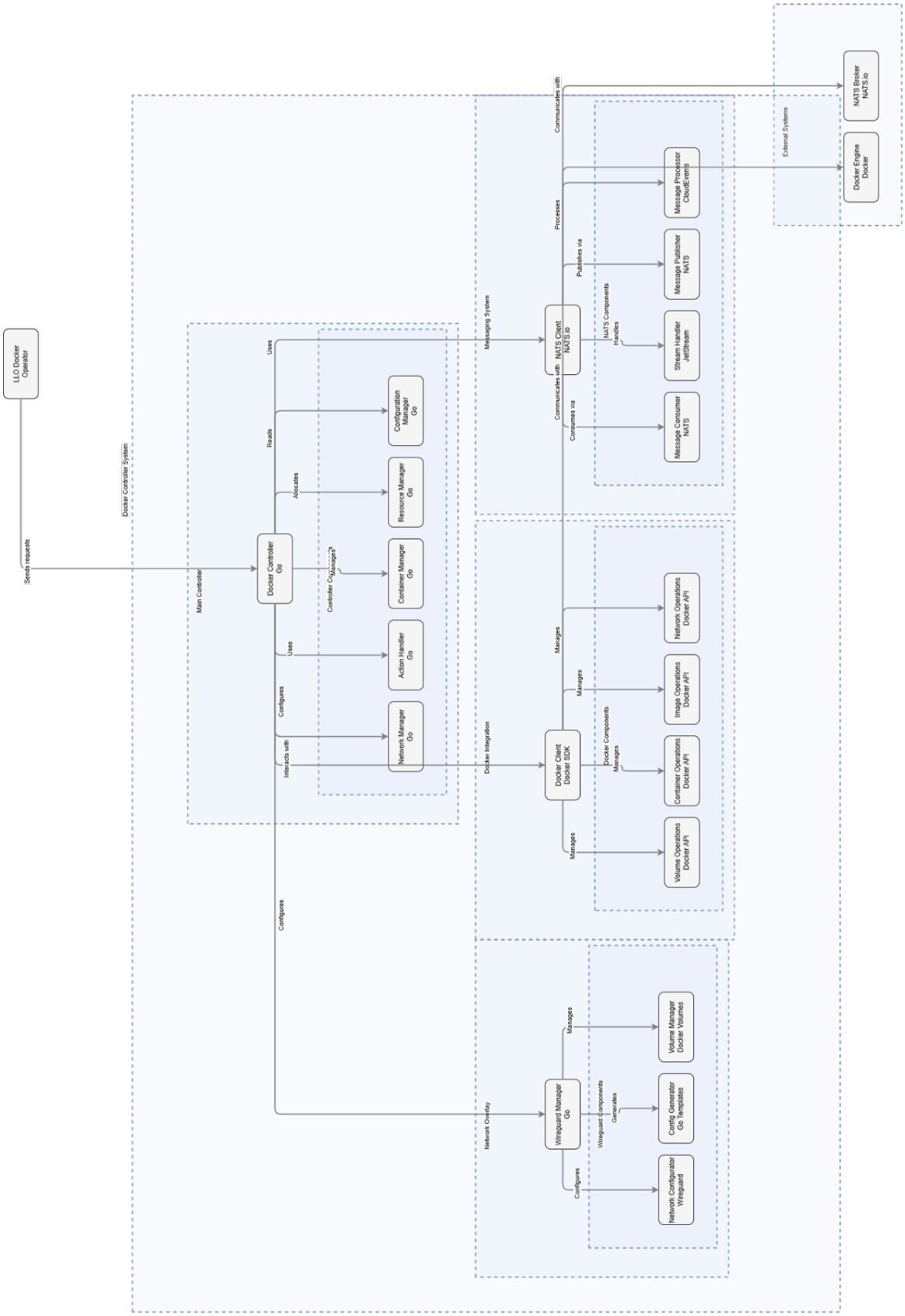


Figura A.4: Diagrama del controlador del LLO de Docker