uc3m | Universidad Carlos III de Madrid

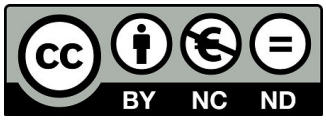
Grado Universitario en Ingeniería Informática
2025-2026

Trabajo Fin de Grado

"Evaluación de frameworks de contenedores virtuales en entornos edge-to-cloud y serverless"

Iván Sebastián Loor Weir

Tutor/es
Dante Domizzi Sánchez Gallegos



Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento - No Comercial - Sin Obra Derivada**

RESUMEN

En este Trabajo de Fin de Grado (TFG), se aborda el desarrollo e implementación de cuatro alternativas tecnológicas destinadas a optimizar el despliegue y la ejecución de aplicaciones en entornos de Internet de las Cosas (IoT, por sus siglas en inglés) y de alta computación, basados en contenedores.

El objetivo es analizar y comparar el comportamiento de diferentes tecnologías de virtualización y orquestación, centrándose en Docker, Kubernetes (específicamente k3s y k0s), Nomad y Apptainer, herramientas orientadas a la ejecución eficiente de aplicaciones en contenedores.

Todo surge de la necesidad de contar con una plataforma flexible que permita evaluar la ejecución de flujos de trabajo bajo distintas configuraciones, estudiando factores como el tiempo de respuesta y la eficiencia en el uso de los recursos.

El proyecto se desarrolla con un enfoque experimental, pasando por pruebas tanto locales como remotas. Se utiliza DAGonStar, un motor de flujos de trabajo basado en grafos acíclicos dirigidos, que facilita la definición y la posterior ejecución de tareas en diferentes entornos. A partir de esto, se definen distintos escenarios de prueba para comparar la ejecución de las mismas tareas en las herramientas mencionadas.

Con base en esto, se obtienen métricas de rendimiento que sirven para extraer conclusiones sobre la idoneidad de cada tecnología según el contexto de uso. Para ello, se realiza un estudio de las herramientas involucradas, de su funcionamiento real y de su integración en un marco común de evaluación, con el fin de aportar una visión práctica sobre el despliegue de aplicaciones en entornos distribuidos.

Palabras clave: Docker; Kubernetes; Apptainer; Nomad; Contenedores; DAGonStar; Entornos distribuidos.

DEDICATORIA

En primer lugar, quiero agradecer a Dios, por guiarme y darme la fortaleza necesaria para completar este proyecto. Solo Él y yo conocemos todas las noches de desvelo, esfuerzo y dudas que han formado parte de este proceso, y sin su ayuda nada de esto habría sido posible.

A mi familia, especialmente a mi padre, mi madre y mi hermano, por su amor incondicional, su comprensión y la confianza que siempre han depositado en mí. Gracias por apoyarme incluso en los momentos más difíciles y por estar siempre presentes, acompañándome en cada paso. Haría falta otra tesis más para acercarme a expresar cuánto los quiero y lo agradecido que estoy por todo lo que hacen por mí.

De manera muy especial, quiero agradecer a mi novia, por su apoyo constante e incondicional a lo largo de todo este proceso y por ser un pilar fundamental en mi día a día. Gracias por tu paciencia, tu ánimo y por estar a mi lado incluso cuando el cansancio y el estrés se hacían notar.

Por último, quiero agradecer a mi tutor, Dante, por su consejo y orientación durante la realización de este trabajo. Su conocimiento y disposición han sido clave para poder llevar esta tesis a buen término.

ÍNDICE GENERAL

1. INTRODUCCIÓN. 1
1.1. Motivación del Trabajo 1
1.2. Objetivos 2
1.3. Recursos Utilizados 2
1.4. Estructura del documento 4
2. ESTADO DEL ARTE 5
2.1. Contexto 5
2.1.1. Computación Edge-to-Cloud 5
2.1.2. Paradigma Serverless y FaaS 6
2.1.3. Virtualización ligera y contenedores 7
2.1.4. Workflows científicos en entornos distribuidos 7
2.2. DAGonStar como Motor de Flujos de Trabajo 8
2.2.1. Visión general y filosofía de diseño 8
2.2.2. Modelo de tareas y extensibilidad 9
2.2.3. Gestión de datos y necesidad de nuevos mecanismos 9
3. DISEÑO DE LA SOLUCIÓN 11
3.1. Visión General de la Arquitectura 11
3.2. Organización del Repositorio y Estructura de Código 12
3.3. Modelo de Clases y Abstracción de Tareas 13
3.3.1. Jerarquía de tareas 13
3.3.2. Diseño de `ApptainerTask` y `RemoteApptainerTask` 15
3.3.3. Diseño de `DockerTask` y `RemoteDockerTask` 18
3.3.4. Diseño de `KubernetesTask` y `RemoteKubernetesTask` 21
3.3.5. Diseño de `NomadTask` y `RemoteNomadTask` 23
3.3.6. Validación funcional inicial: Scripts de Taskflow y Dataflow 26
3.4. Gestión de Credenciales y Parámetros: `dagon.ini` 29
3.5. Diseño del Mecanismo de *Staging* Transversal 30
3.5.1. Data Mover y operaciones básicas 30

3.5.2. Staging para Apptainer, Docker y Nomad 31
3.5.3. Staging para Kubernetes: *Streaming* entre pods 32
3.6. Módulo de Monitorización y Trazabilidad 32
3.7. Síntesis de las Decisiones de Diseño 33
4. EVALUACIÓN 34
4.1. Metodología de Evaluación 34
4.1.1. Escenario experimental 34
4.1.2. Infraestructura hardware y software 36
4.1.3. Configuración y despliegue de los entornos de ejecución 38
4.1.4. Métricas y herramientas 43
4.1.5. Tipología de experimentos 46
4.1.6. Implementación de Benchmarks con DAGonStar 50
4.1.7. Implementación y Configuración del Workflow IoT 54
4.1.8. Gestión de Datos y Monitorización en MongoDB Atlas 63
4.2. Configuración de la Infraestructura de Monitorización 68
4.2.1. Monitorización en el Nodo Edge (Raspberry Pi 5) 69
4.2.2. Monitorización en el Nodo Fog (PC) 70
4.2.3. Ingestión y Visualización: Prometheus y Grafana 70
4.3. Evaluación del Funcionamiento 71
4.3.1. Verificación mediante pruebas unitarias 72
4.3.2. Validación del flujo Fog–Edge–Cloud 72
4.3.3. Robustez y manejo de errores 73
4.4. Evaluación del Rendimiento 73
4.4.1. Benchmark de Nodo Único (Edge) 73
4.4.2. Benchmark Distribuido (Edge–Fog) 76
4.4.3. Síntesis de Resultados de Benchmarks 78
4.4.4. Workflow de Telemetría IoT Completo (Edge–Fog–Cloud) 79
4.4.5. Visualización de Métricas en Tiempo Real: Paneles de Grafana (Workflow IoT) 81
4.4.6. Análisis de Consumo Energético (Workflow IoT) 86
4.4.7. Síntesis Final: Eficiencia Energética, Overhead y Sostenibilidad 89

4.4.8. Limitaciones técnicas de Kubernetes en el Edge: el caso de k3s y k0s . . . 92
5. PLANIFICACIÓN, PRESUPUESTO Y MARCO REGULATORIO 93
5.1. Planificación 93
5.2. Estructuración 94
5.3. Entorno Socioeconómico 95
5.4. Presupuesto 96
5.4.1. Costes de personal 96
5.4.2. Costes materiales y amortizaciones 97
5.4.3. Costes Generales 97
5.4.4. Escenario de referencia con costes indirectos e IVA 98
5.5. Impacto 98
5.6. Contribución a los Objetivos de Desarrollo Sostenible 99
5.7. Matriz DAFO 100
5.8. Marco Regulatorio 100
6. CONCLUSIONES Y TRABAJO FUTURO 102
6.1. Conclusiones 102
6.2. Trabajo Futuro 103
BIBLIOGRAFÍA 105
ANEXOS 1
I. REPOSITORIO DE CÓDIGO EN GITHUB 1
II. DECLARACIÓN DE USO DE IA GENERATIVA 2

ÍNDICE DE FIGURAS

2.1 Representación de los niveles de infraestructura Edge-to-Cloud, distancias típicas y latencias asociadas 6

2.2 Visualización de un Grafo Acíclico Dirigido (DAG) representativo de un workflow de contenedores 8

3.1 Arquitectura general de DAGonStar y sus ejecutores. 12

3.2 Diagrama de clases de las tareas desplegadas en contenedores en DAGonStar. 14

3.3 Esquema del componente *Stager* y sus operaciones de *staging*. 31

3.4 Mecanismo de *staging* basado en volúmenes para Docker, Apptainer y Nomad. 31

3.5 Mecanismo de *staging* basado en *streaming* entre pods para Kubernetes. . 32

4.1 Montaje físico del nodo Edge: Raspberry Pi 5 conectada a Arduino con sensor DHT11 35

4.2 Diagrama de secuencia del workflow Fog–Edge–Cloud. 35

4.3 Diagrama de objetos del workflow. 36

4.4 Diagrama de actividad del workflow IoT con DAGonStar. 55

4.5 Dashboard de MongoDB Atlas 64

4.6 Métricas de rendimiento del cluster en MongoDB Atlas 64

4.7 Colecciones de métricas en MongoDB Compass 65

4.8 Vista de documentos en MongoDB Compass 65

4.9 Panel de Grafana: Potencia (Nomad) 82

4.10 Panel de Grafana: CPU (Nomad) 82

4.11 Panel de Grafana: RAM (Nomad) 82

4.12 Panel de Grafana: Temperatura (Nomad) 83

4.13 Panel de Grafana: Potencia (Docker) 83

4.14 Panel de Grafana: CPU (Docker) 84

4.15 Panel de Grafana: RAM (Docker) 84

4.16 Panel de Grafana: Temperatura (Docker) 84

4.17 Panel de Grafana: Potencia (Apptainer) 85
4.18 Panel de Grafana: CPU (Apptainer) 85
4.19 Panel de Grafana: RAM (Apptainer) 86
4.20 Panel de Grafana: Temperatura (Apptainer) 86
4.21 Perfiles de consumo energético con Nomad. 87
4.22 Perfiles de consumo energético con Docker. 88
4.23 Perfiles de consumo energético con Apptainer. 89

5.1 Diagrama de Gantt del proyecto. 94

ÍNDICE DE TABLAS

4.1 Resultados de ejecución Edge-Only (Nomad en RPi 5) 74

4.2 Resultados de ejecución Edge-Only (Docker en RPi 5) 74

4.3 Resultados de ejecución Edge-Only (Apptainer en RPi 5) 75

4.4 Impacto comparativo en el almacenamiento (Escenario Edge-Only). . . . 75

4.5 Resultados de ejecución híbrida Edge-Fog (Nomad) 76

4.6 Resultados de ejecución híbrida Edge-Fog (Docker) 76

4.7 Resultados de ejecución híbrida Edge-Fog (Apptainer) 77

4.8 Impacto comparativo en el almacenamiento (Escenario Edge-Fog). 77

4.9 Métricas del Workflow IoT con Nomad (Edge–Fog–Cloud). 79

4.10 Métricas del Workflow IoT con Docker (Edge–Fog–Cloud). 80

4.11 Métricas del Workflow IoT con Apptainer (Edge–Fog–Cloud). 80

4.12 Métricas energéticas medias con Nomad. 87

4.13 Métricas energéticas medias con Docker. 88

4.14 Métricas energéticas medias con Apptainer. 89

4.15 Comparativa global de eficiencia normalizada, impacto térmico y overhead de orquestación. 90

5.1 Fases de planificación y esfuerzo estimado. 93

5.2 Coste de personal. 96

5.3 Costes directos de material. 97

5.4 Costes totales del proyecto. 98

5.5 Matriz DAFO del proyecto. 100

1. INTRODUCCIÓN

La computación ha experimentado en los últimos años una transición desde modelos centralizados, apoyados exclusivamente en grandes centros de datos, hacia arquitecturas distribuidas que combinan recursos cercanos al usuario con infraestructuras cloud de gran escala. En este contexto surgen los entornos de computación *Edge-to-Cloud* y *Serverless*, un paradigma que integra de forma coherente los niveles *Edge*, *Fog* y *Cloud*, permitiendo decidir dinámicamente dónde ejecutar cada carga de trabajo en función de requisitos de latencia, consumo energético o capacidad de cómputo [1], [2]. Dentro de este escenario, los motores de *workflow* científico juegan un papel clave, ya que proporcionan un marco estructurado para definir, orquestar y monitorizar experimentos complejos sobre infraestructuras heterogéneas.

1.1. Motivación del Trabajo

La creciente proliferación de dispositivos IoT, sensores y aplicaciones de análisis en tiempo real ha impulsado la necesidad de procesar datos cerca de su fuente, reduciendo la dependencia exclusiva de la nube y mitigando los cuellos de botella de comunicación. Sin embargo, esta evolución hacia entornos distribuidos *Edge-to-Cloud* introduce una fuerte heterogeneidad en términos de arquitecturas hardware (x86_64, ARMv8), sistemas operativos, redes y tecnologías de despliegue, lo que complica la ejecución reproducible de aplicaciones científicas y de misión crítica [3], [4].

En particular, la ejecución de flujos de trabajo distribuidos se enfrenta a tres desafíos principales. En primer lugar, el **aislamiento** de dependencias: distintas tareas pueden requerir versiones incompatibles de librerías o herramientas, lo que dificulta su instalación directa en los nodos físicos. En segundo lugar, la **portabilidad**: una tarea desarrollada inicialmente en un entorno de escritorio debe poder ejecutarse sin modificaciones en nodos Edge de recursos limitados o en clústeres Cloud con sistemas de colas y contenedores. En tercer lugar, la **gestión de datos**: el movimiento de ficheros intermedios entre Edge, Fog y Cloud es propenso a errores, especialmente cuando no existe un sistema de archivos compartido entre todos los nodos.

Los motores de workflow tradicionales han sido diseñados principalmente para entornos homogéneos o para infraestructuras donde se dispone de un sistema de almacenamiento compartido. Esto limita su aplicación en entornos *Edge-to-Cloud*, en el que las tareas deben coordinarse a través de conexiones de red potencialmente inestables y con distintos mecanismos de autenticación y acceso. En este contexto, **DAGonStar** destaca por su filosofía de "ejecutar en cualquier parte", pero requiere una integración más profunda con tecnologías de contenedores y mecanismos de *staging* de datos para explotar plenamente las oportunidades de este tipo de infraestructuras [1], [2]. Esta necesidad

constituye la motivación central del presente Trabajo Fin de Grado.

1.2. Objetivos

El objetivo general de este trabajo es extender el motor de workflows DAGonStar con capacidades avanzadas para la ejecución en contenedores virtuales y gestión de datos que permitan ejecutar flujos de trabajo científicos de forma transparente y reproducible en entornos *Edge-to-Cloud* y *Serverless*. Para concretar este propósito, se definen los siguientes objetivos específicos:

- **Diseño e implementación de nuevas clases para la generación de tareas en contenedores virtuales.** Desarrollar e integrar en DAGonStar componentes como `ApptainerTask`, `DockerTask`, `NomadTask` y `KubernetesTask` [5], [6], [7], [8], junto con sus variantes remotas, de forma que el usuario pueda seleccionar el tipo de ejecutor manteniendo una interfaz homogénea.
- **Desarrollo de un mecanismo de *staging* transversal.** Diseñar un subsistema de movimiento de datos que permita intercambiar ficheros entre tareas ejecutadas en contenedores y nodos heterogéneos, combinando áreas compartidas en el host, transferencia segura mediante SSH [9]/SCP [10] y técnicas de *streaming* entre *pods* de Kubernetes.
- **Integración de monitorización de rendimiento y energía.** Incorporar un sistema de métricas basado en Prometheus y Grafana [11], [12] que recoja información sobre tiempos de ejecución, uso de recursos y consumo energético, con especial atención al comportamiento de dispositivos Edge como la Raspberry Pi.
- **Evaluación experimental en un escenario Fog–Edge–Cloud.** Diseñar y ejecutar un caso de estudio representativo, basado en la captura y procesamiento de datos de sensores, que permita comparar el comportamiento de las distintas tecnologías de contenedores y validar la utilidad de la solución propuesta en términos de rendimiento, portabilidad y eficiencia energética.

El cumplimiento de estos objetivos permitirá valorar hasta qué punto la extensión de DAGonStar facilita la orquestación de workflows en estos entornos y qué compromisos se introducen en términos de complejidad, tiempos de despliegue y sobrecoste de monitorización.

1.3. Recursos Utilizados

Para el desarrollo del proyecto se han empleado diversos recursos hardware y software, seleccionados para construir un prototipo funcional del entorno *Edge-to-Cloud*. El

diseño permite orquestar tareas desde un nodo central hacia dispositivos en el extremo, persistiendo finalmente los resultados en la nube.

Desde el punto de vista hardware, la infraestructura se divide en tres niveles:

- **Nivel Fog y Orquestación**: un PC de escritorio que actúa como punto de orquestación central. Desde este nodo se ejecuta el motor de *workflows* y se gestiona el ciclo de vida de las tareas mediante conexiones SSH remotas, enviando órdenes de ejecución a los dispositivos en el borde.
- **Nivel Edge**: una **Raspberry Pi 5** [13] configurada como nodo de computación en el extremo. Este dispositivo interactúa con sensores físicos (temperatura y humedad) y ejecuta los diversos motores de contenedores evaluados.
- **Nivel Cloud**: un clúster de base de datos gestionado en la nube (**MongoDB Atlas**), que centraliza la persistencia de las métricas y resultados, asegurando la accesibilidad global y la trazabilidad de los experimentos.

En cuanto al software, los recursos más relevantes son:

- El motor de workflows **DAGonStar**, sobre el que se implementan las nuevas capacidades de ejecución remota y mediante contenedores virtuales.
- Los sistemas de contenedores **Docker** [6] y **Apptainer** [5], junto con el orquestador **HashiCorp Nomad** [7] y las distribuciones ligeras **k3s** y **k0s** [8], [14], que proporcionan los entornos de ejecución en el Edge y el Fog.
- El stack de monitorización compuesto por **Prometheus** [11], **Node Exporter** y **Grafana** [12], utilizado para la captura y visualización dinámica de métricas.
- Herramientas de comunicación y transferencia como `ssh`, `scp` [15] y la API de **MongoDB Atlas** [16].
- El entorno de desarrollo integrado **Visual Studio Code**[17], utilizado para la codificación y depuración de los componentes en **Python** [18].
- Las herramientas de modelado y diseño **Draw.io / diagrams.net**[19] y **PlantUML**[20], empleadas para la elaboración de los diversos diagramas de arquitectura, clases y actividad presentes en esta memoria.

Todo el código desarrollado se ha gestionado mediante el sistema de control de versiones **Git**[21], manteniendo un repositorio público en **GitHub**[22] que contiene la implementación de las nuevas clases de tareas, los scripts de despliegue y los ficheros de configuración utilizados en los experimentos. Esta organización facilita la reproducibilidad del trabajo y sienta las bases para su futura extensión en el marco de proyectos de investigación relacionados.

1.4. Estructura del documento

El presente documento se organiza en seis capítulos principales que guían al lector a través de la investigación y desarrollo realizados:

- El **Capítulo 1 (Introducción)** presenta la motivación del trabajo, los objetivos perseguidos, los recursos empleados y la estructura de la memoria.
- El **Capítulo 2 (Estado del Arte)** describe el contexto teórico de la computación *Edge-to-Cloud*, las tecnologías de contenedores virtuales y los fundamentos del motor de *workflows* **DAGonStar**.
- El **Capítulo 3 (Diseño de la Solución)** detalla la arquitectura propuesta, la jerarquía de las nuevas clases de tareas y el diseño del mecanismo de *staging* transversal para la gestión de datos.
- El **Capítulo 4 (Evaluación Experimental)** explica la configuración del entorno de pruebas, el stack de monitorización desplegado y analiza los resultados obtenidos en términos de rendimiento y eficiencia energética.
- El **Capítulo 5 (Planificación, Presupuesto y Marco Regulatorio)** expone el calendario de trabajo seguido, el análisis detallado de los costes asociados, el estudio del impacto socioeconómico (incluyendo los Objetivos de Desarrollo Sostenible) y el análisis del marco legal, ético y normativo aplicable al proyecto.
- El **Capítulo 6 (Conclusiones y Trabajo Futuro)** resume los logros alcanzados, las lecciones aprendidas y propone diversas líneas para la futura extensión de la solución.

2. ESTADO DEL ARTE

En este capítulo se presenta el contexto teórico y tecnológico en el que se enmarca el Trabajo Fin de Grado. En primer lugar, se describen los principales conceptos asociados a la computación *Edge-to-Cloud* y al uso de contenedores virtuales en entornos distribuidos. A continuación, se introduce el motor de *workflows* DAGonStar, poniendo el foco en aquellas características que resultan más relevantes para su extensión hacia escenarios Fog–Edge–Cloud. El objetivo es disponer de una base conceptual sólida que justifique las decisiones de diseño adoptadas en los capítulos posteriores.[1], [3], [4]

2.1. Contexto

2.1.1. Computación Edge-to-Cloud

La computación *Edge-to-Cloud* hace referencia a un modelo de infraestructura en el que los recursos de cómputo y almacenamiento se distribuyen de forma continua desde dispositivos muy cercanos a la fuente de datos (*Edge*) hasta grandes centros de datos en la nube (*Cloud*), pasando por capas intermedias conocidas como *Fog*.[3], [4] Este enfoque permite situar cada tarea en el nivel de la jerarquía que mejor se adapta a sus necesidades de latencia, capacidad de procesamiento, consumo energético o requisitos de privacidad.

En el **Edge** se sitúan dispositivos con recursos limitados (p. ej., Raspberry Pi, placas Jetson o gateways IoT) que ejecutan tareas próximas al sensor y requieren respuestas casi en tiempo real. La capa **Fog** agrupa nodos con mayor capacidad que el Edge, a menudo desplegados en microcentros de datos o infraestructuras de proximidad, que actúan como puntos de agregación, filtrado y preprocesamiento de la información. Por último, la capa **Cloud** ofrece recursos virtualmente elásticos y almacenamiento masivo, idóneos para tareas de cómputo intensivo, análisis histórico y entrenamiento de modelos de inteligencia artificial.[3], [4], [23]

Como se ilustra en la Figura 2.1, la transición desde el dispositivo hasta la nube implica un incremento progresivo en la distancia típica (desde menos de 1 km hasta más de 1000 km) y en la latencia media (desde 1 ms hasta valores superiores a los 20 ms). Este gradiente tecnológico define los límites operativos de cada nivel y justifica la necesidad de una orquestación inteligente que aproveche la cercanía del Edge para tareas de baja latencia y la potencia del Cloud para el procesamiento masivo a largo plazo [24].

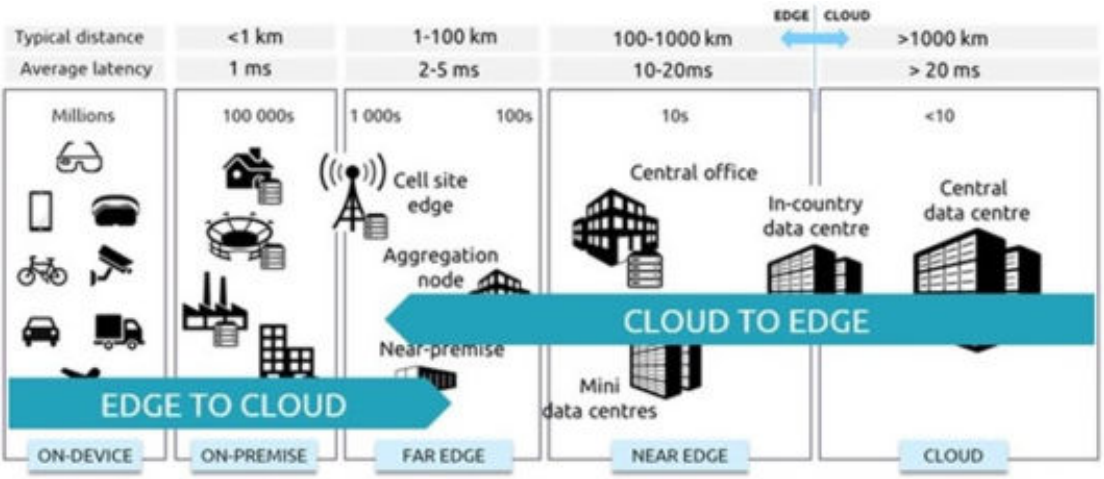


Figura 2.1
Representación de los niveles de infraestructura Edge-to-Cloud, distancias típicas y latencias asociadas [24].

2.1.2. Paradigma Serverless y FaaS

El paradigma *serverless* (o computación sin servidor) representa una evolución del modelo de nube en el que la infraestructura subyacente queda completamente abstraída para el desarrollador. En este modelo, la gestión de servidores, el escalado de recursos y el aprovisionamiento de capacidad recaen totalmente en la plataforma, permitiendo centrar el esfuerzo únicamente en la lógica de negocio.[25]

Dentro de este ecosistema, el concepto de **Function-as-a-Service (FaaS) [26]** desempeña un papel fundamental. Las aplicaciones se estructuran como un conjunto de funciones independientes y efímeras que se ejecutan en respuesta a eventos concretos (p. ej., la llegada de un dato de un sensor o una petición HTTP). Esta naturaleza reactiva y el modelo de facturación basado en el uso real hacen que el enfoque *serverless* sea altamente eficiente para tareas esporádicas o con patrones de carga variables.[25]

En el contexto de los entornos *Edge-to-Cloud* abordados en esta tesis, el modelo *serverless* se perfila como una solución estratégica para optimizar el uso de nodos con recursos limitados. Al emplear tecnologías de contenedores para encapsular estas funciones, es posible desplegar capacidades de procesamiento bajo demanda en el borde de la red, garantizando que el consumo energético y de memoria se limite estrictamente al tiempo de ejecución de la tarea. Esta aproximación facilita la creación de *workflows* ágiles y sostenibles, donde la infraestructura escala dinámicamente desde el Edge hasta el Cloud en función de la demanda del flujo de trabajo.

2.1.3. Virtualización ligera y contenedores

La heterogeneidad de estos entornos hace necesario disponer de mecanismos que faciliten la portabilidad y el aislamiento de aplicaciones. En este contexto, las tecnologías de contenedores han adquirido un papel central, al permitir empaquetar una aplicación junto con sus dependencias en una imagen autocontenida que puede ejecutarse, con cambios mínimos, en distintas plataformas.[5], [8]

En el contexto de esta tesis, se define un **framework de contenedores** como el ecosistema tecnológico que integra tanto el motor de ejecución (*runtime*) como las herramientas de gestión, orquestación y abstracción de recursos. Mientras que los *runtimes* se ocupan del aislamiento a nivel de sistema operativo, el *framework* proporciona la lógica necesaria para coordinar múltiples instancias, gestionar su ciclo de vida y facilitar la comunicación en entornos altamente distribuidos.

Docker popularizó el uso de contenedores mediante el aprovechamiento de primitivas del kernel Linux como *cgroups* y *namespaces*, ofreciendo un modelo de despliegue muy atractivo para desarrolladores y operadores de plataformas cloud.[6] No obstante, su modelo de seguridad basado en demonios privilegiados plantea limitaciones en entornos multiusuario como HPC [27] o infraestructuras académicas compartidas. Como respuesta a estas necesidades surgió Apptainer (antes Singularity), específicamente orientado a cargas de trabajo científicas, que ejecuta los procesos con los privilegios del usuario sin requerir acceso *root* y emplea imágenes inmutables en formato SIF, favoreciendo la reproducibilidad a largo plazo.[5]

Sobre estas tecnologías se sitúan los orquestadores de contenedores, entre los cuales Kubernetes se ha consolidado como estándar de facto. Para el ámbito de la computación distribuida en el borde resultan especialmente relevantes las variantes ligeras como **k3s** y **k0s** [8], [14], que reducen la huella de recursos manteniendo funcionalidades clave de autoescalado, *self-healing* y gestión del estado. Asimismo, orquestadores como HashiCorp Nomad [7] emergen como soluciones alternativas por su simplicidad y capacidad para gestionar tanto contenedores como procesos nativos en infraestructuras heterogéneas. En conjunto, la combinación de contenedores y orquestadores proporciona los bloques fundamentales sobre los que construir soluciones de orquestación de *workflows* en entornos Edge–Fog–Cloud.

2.1.4. Workflows científicos en entornos distribuidos

Los *workflows* científicos constituyen una herramienta esencial para estructurar experimentos complejos como grafos de tareas, facilitando la automatización, trazabilidad y repetición de procesos de análisis de datos o simulación numérica.[1], [3] Tradicionalmente, motores como Pegasus [28], Kepler [29] o Airflow [30] se han desplegado sobre infraestructuras relativamente homogéneas, en las que suele existir un sistema de archivos compartido o una cola de trabajos común que simplifica el intercambio de datos entre

tareas.

Cuando los *workflows* se extienden a entornos *Edge-to-Cloud* aparecen nuevos retos: la ausencia de almacenamiento compartido entre todos los nodos, la variabilidad de la conectividad de red, la coexistencia de distintos *runtimes* de contenedores y la necesidad de gestionar datos que se mueven entre Edge, Fog y Cloud.[4] Estas limitaciones han motivado propuestas recientes que abordan la comunicación a través de canales abstractos o modelos de *staging* explícito, en los que el motor de *workflow* asume responsabilidades adicionales en la orquestación del movimiento de datos. El trabajo presentado en esta memoria se inscribe precisamente en esta línea, tomando como base el motor DAGonStar [1], [2].

2.2. DAGonStar como Motor de Flujos de Trabajo

2.2.1. Visión general y filosofía de diseño

DAGonStar (*Directed Acyclic Graph on Star*) es un motor de *workflows* cuyo objetivo principal es ejecutar grafos acíclicos dirigidos como colecciones de trabajos paralelos sobre infraestructuras heterogéneas [1], [2]. Un **Grafo Acíclico Dirigido (DAG)** es una estructura compuesta por nodos (tareas) y aristas direccionales (dependencias) que no contiene ciclos, lo que garantiza un flujo de ejecución unidireccional y predecible. En DAGonStar, el modelo de programación se basa en la definición de estas tareas (`Task`) y sus dependencias, de forma que cada nodo del grafo representa un trabajo independiente y cada arista captura relaciones de precedencia o intercambio de datos.

La Figura 2.2 ayuda a visualizar cómo actúan estos flujos de trabajo en un entorno de contenedores. En esta representación, cada nodo puede simbolizar una tarea desplegada en un contenedor (p. ej., una captura de sensor o un filtrado de datos) que depende de la salida de procesos previos. Este esquema es fundamental para entender la experimentación realizada en el Capítulo 4, donde se orquestan flujos de telemetría IoT reales siguiendo una lógica similar.

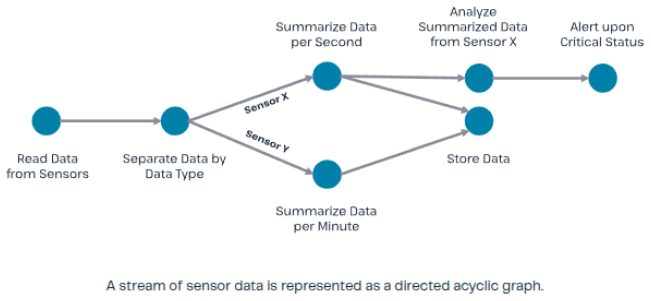


Figura 2.2
Visualización de un Grafo Acíclico Dirigido (DAG) representativo de un workflow de contenedores [31].

La arquitectura sigue un patrón de estrella (*star*), en el que un orquestador central coordina la ejecución de las tareas, mientras que el trabajo efectivo se realiza en nodos externos que pueden pertenecer a distintas infraestructuras (equipos locales, clústeres HPC [27], instancias cloud o contenedores).[1] Esta separación entre la lógica de control y la ejecución remota facilita la integración con nuevos tipos de recursos mediante la implementación de clases de tareas especializadas.

2.2.2. Modelo de tareas y extensibilidad

En **DAGonStar**, las tareas se definen como clases Python que implementan una interfaz común basada en métodos como `on_execute()` y `on_garbage()`, responsables respectivamente de lanzar el trabajo y realizar la limpieza de recursos al finalizar [1], [2]. A partir de esta interfaz se han desarrollado diferentes familias de tareas, p. ej., tareas nativas (ejecución local), tareas *batch* [32] (integradas con gestores de colas como SLURM [33]) o tareas remotas (basadas en SSH). Esta organización se refleja en el diagrama de clases incluido en el capítulo de diseño, donde se muestra la jerarquía de herencia y el uso de polimorfismo para encapsular comportamientos específicos.

La filosofía de extensibilidad de DAGonStar resulta especialmente adecuada para el presente trabajo, ya que permite incorporar nuevas clases asociadas a contenedores sin alterar el núcleo del motor. En concreto, la tesis propone las clases `ApptainerTask`, `DockerTask`, `NomadTask` y `KubernetesTask`, además de sus variantes remotas, que respetan la interfaz estándar de tareas pero delegan la ejecución en los *runtimes* de contenedores correspondientes.[1] De esta forma, los usuarios pueden seguir describiendo los *workflows* con el mismo modelo conceptual, mientras que la decisión sobre dónde y cómo se ejecuta cada tarea se abstrae en la configuración de la propia tarea.

2.2.3. Gestión de datos y necesidad de nuevos mecanismos

Un aspecto clave en DAGonStar es la gestión del movimiento de datos entre tareas. En escenarios tradicionales, el motor puede asumir que las tareas comparten un sistema de archivos común o que el intercambio de ficheros se realiza mediante rutas accesibles desde todos los nodos.[1] Sin embargo, en estos escenarios *Edge-to-Cloud* esta suposición no se cumple: los dispositivos Edge suelen disponer únicamente de almacenamiento local, los nodos Fog pueden estar en redes distintas y el Cloud introduce sus propios mecanismos de volúmenes y almacenamiento persistente.

Estas restricciones hacen necesario dotar a DAGonStar de un mecanismo explícito de *staging* transversal que maneje la transferencia de datos entre infraestructuras heterogéneas, de modo que una tarea que se ejecuta en un contenedor Apptainer en una Raspberry Pi pueda producir un fichero que, posteriormente, consuma una tarea desplegada en un *pod* de Kubernetes en el Cloud. La propuesta desarrollada en esta tesis aborda esta necesidad mediante un componente de *Data Mover* integrado en el motor, capaz de

combinar copias basadas en SSH/SCP, directorios de intercambio en el host y técnicas de *streaming* entre contenedores.[3], [4]

En conclusión, DAGonStar proporciona una base flexible y extensible sobre la que construir soluciones de orquestación de *workflows* en infraestructuras *Edge-to-Cloud* y *Serverless*. No obstante, para alcanzar este objetivo resulta imprescindible incorporar de forma nativa el soporte a contenedores y la gestión avanzada de datos que se describen en el capítulo de diseño de la solución.

3. DISEÑO DE LA SOLUCIÓN

En este capítulo se describe la arquitectura propuesta para extender DAGonStar [1] con soporte avanzado de contenedores y un mecanismo de *staging* transversal orientado a entornos *Edge-to-Cloud*. El contenido se organiza en varios apartados: en primer lugar se presenta la visión general del sistema; a continuación se detalla el diseño de las nuevas clases para la ejecución de tareas desplegadas en contenedores; se explica la gestión de configuraciones y credenciales mediante el archivo `dagon.ini`; posteriormente se describe el subsistema de *staging* y movimiento de datos; por último, se introduce el módulo de monitorización y se sintetizan las principales decisiones de diseño.

3.1. Visión General de la Arquitectura

La solución propuesta sigue una arquitectura en capas que separa claramente la definición del *workflow*, la lógica de ejecución de tareas y los servicios auxiliares de datos, credenciales y monitorización. En la capa superior se sitúa la aplicación DagOn* y las herramientas de gestión (interfaz web, API REST), que permiten al usuario definir y lanzar *workflows* sin necesidad de conocer los detalles de la infraestructura subyacente. En la capa intermedia se encuentran las clases de tareas y el componente de *Stager*, responsables de coordinar la ejecución en contenedores y el movimiento de ficheros intermedios. Finalmente, la capa inferior está formada por los distintos ejecutores (Docker [6], Apptainer [5], Kubernetes [8], colas *batch* [27], [32], SSH) y los recursos físicos Edge, Fog y Cloud.

La Figura 3.1 recoge un esquema de alto nivel de la arquitectura **DAGonStar** [1], [2], donde se observan los distintos tipos de ejecutores soportados y su relación con el motor de *workflows* y con los mecanismos de transferencia de datos. El diseño mantiene la filosofía original de **DAGonStar** de “ejecutar en cualquier parte”, pero incorpora explícitamente la noción de recursos basados en contenedores y cloud, así como un módulo de *Data Mover* configurable.

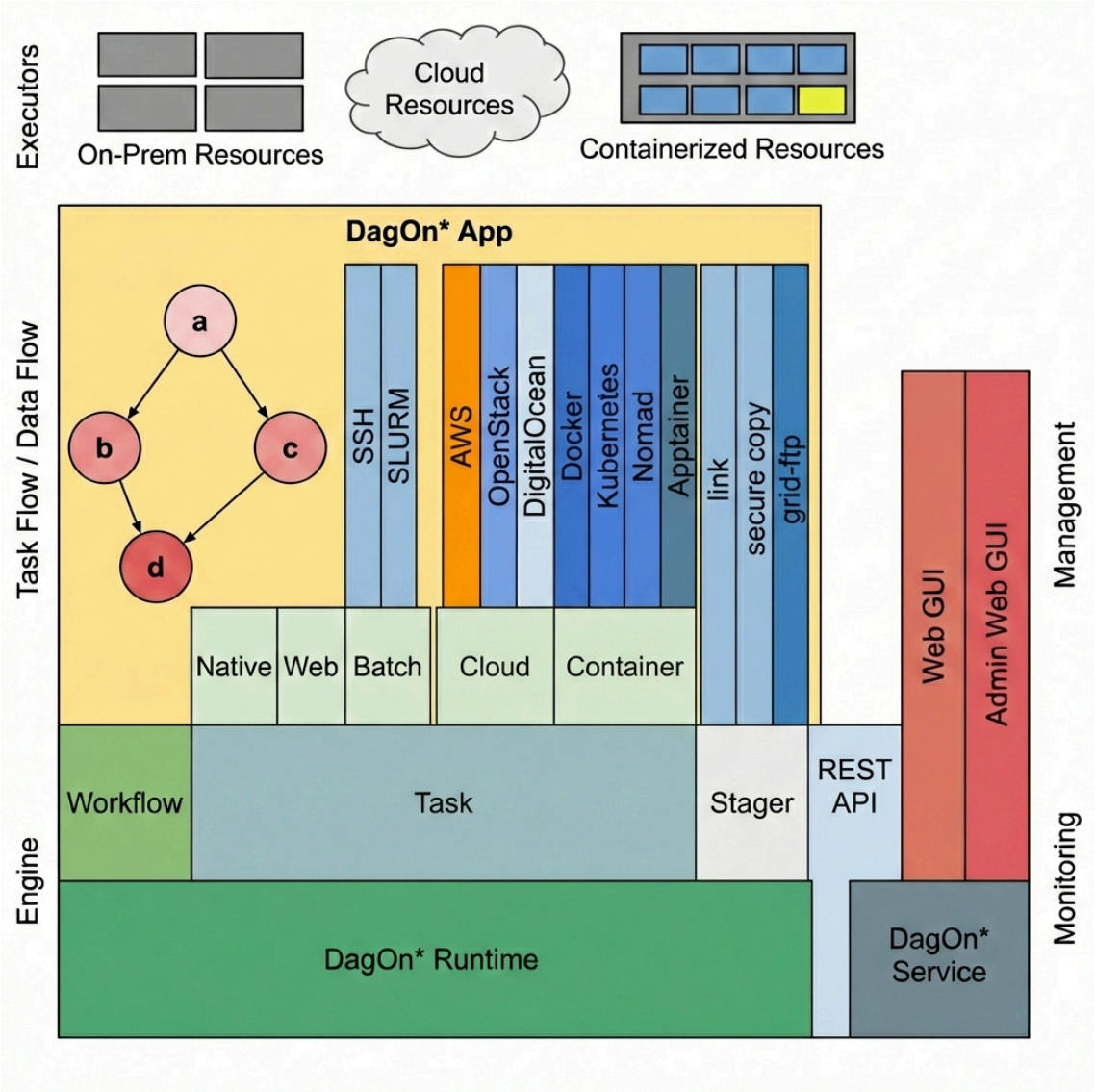


Figura 3.1
Arquitectura general de DAGonStar y sus ejecutores.

3.2. Organización del Repositorio y Estructura de Código

Para facilitar la comprensión de la implementación y asegurar la reproducibilidad de los experimentos, el código fuente se ha organizado siguiendo una estructura modular. El repositorio principal de GitHub [22] contiene tanto el motor del *workflow* como las validaciones y la propia documentación técnica. A continuación se detalla la arquitectura de directorios del proyecto:

```
dagon/                    # Nucleo del framework (Core)
    ├── apptainer_task.py  # Implementacion de Apptainer
    ├── docker_task.py     # Implementacion de Docker
    ├── kubernetes_task.py # Implementacion de Kubernetes
    ├── nomad_task.py      # Implementacion de Nomad
```

```
    ├── remote.py          # Logica de ejecucion SSH
    └── task.py            # Clase base abstracta Task
examples/                  # Scripts de validacion y casos de uso
    ├── taskflow/          # Orquestacion basica local/remota
    ├── dataflow/          # Flujos con movimiento de datos
    └── edge_fog/          # Escenario de evaluacion Raspi+PC
tests/                     # Pruebas unitarias de los componentes
```

CÓDIGO 3.1. Estructura simplificada del repositorio de DAGonStar

Los componentes principales se describen a continuación:

- **Directorio `dagon/`**: Contiene la lógica central del sistema. Aquí es donde se han integrado las nuevas clases para la ejecución de tareas desplegadas en contenedores. Cada módulo (p. ej., `nomad_task.py`) encapsula la complejidad de su respectiva tecnología (drivers, APIs REST, clientes SDK).
- **Directorio `examples/`**: Es el banco de pruebas del proyecto. Se divide en subcategorías que validan desde la conectividad básica (`taskflow`) hasta la interoperabilidad de datos (`dataflow`) y el escenario real de evaluación energética en el Edge (`edge_fog`).
- **Directorio `tests/`**: Incluye la suite de pruebas automatizadas que garantizan que los cambios en el núcleo no degraden la funcionalidad de los ejecutores existentes (HPC [27], Batch [27], [32]).

3.3. Modelo de Clases y Abstracción de Tareas

3.3.1. Jerarquía de tareas

El punto de partida del diseño es el modelo de clases de **DAGonStar** [1], [2]. La Figura 3.2 muestra la jerarquía orientada a objetos empleada para representar las distintas tareas del sistema. La clase abstracta `Task` define la interfaz genérica, incluyendo atributos como el nombre, el comando a ejecutar y los métodos `on_execute()` y `on_garbage()` que implementan la lógica de ejecución y limpieza.

A partir de `Task` se derivan dos ramas principales: `Batch` [32], orientada a ejecuciones secuenciales o integradas con gestores de colas, y `RemoteTask`, que encapsula la ejecución remota mediante SSH. Las nuevas clases diseñadas para tareas desplegadas en contenedores se construyen sobre estas ramas:

- **`ApptainerTask`**: hereda de `Batch` e incorpora atributos específicos como la ruta a la imagen SIF y la configuración de *overlays* de escritura.

- **`DockerTask`**: también heredera de `Batch`, utiliza un cliente Docker para crear contenedores efímeros o persistentes según la configuración.
- **`KubernetesTask`**: extiende `Batch` y emplea la API de Kubernetes para gestionar *pods* y ejecutar comandos dentro de ellos.
- **`NomadTask`**: hereda de `Batch` e interactúa con el orquestador HashiCorp Nomad [7] para el despliegue de trabajos basados en Docker o ejecutables nativos.
- **`RemoteApptainerTask`**, **`RemoteDockerTask`**, **`RemoteKubernetesTask`** y **`RemoteNomadTask`**: combinan la funcionalidad de `RemoteTask` con las respectivas clases especializadas, gestionando el ciclo de vida de tareas que se ejecutan en nodos remotos.

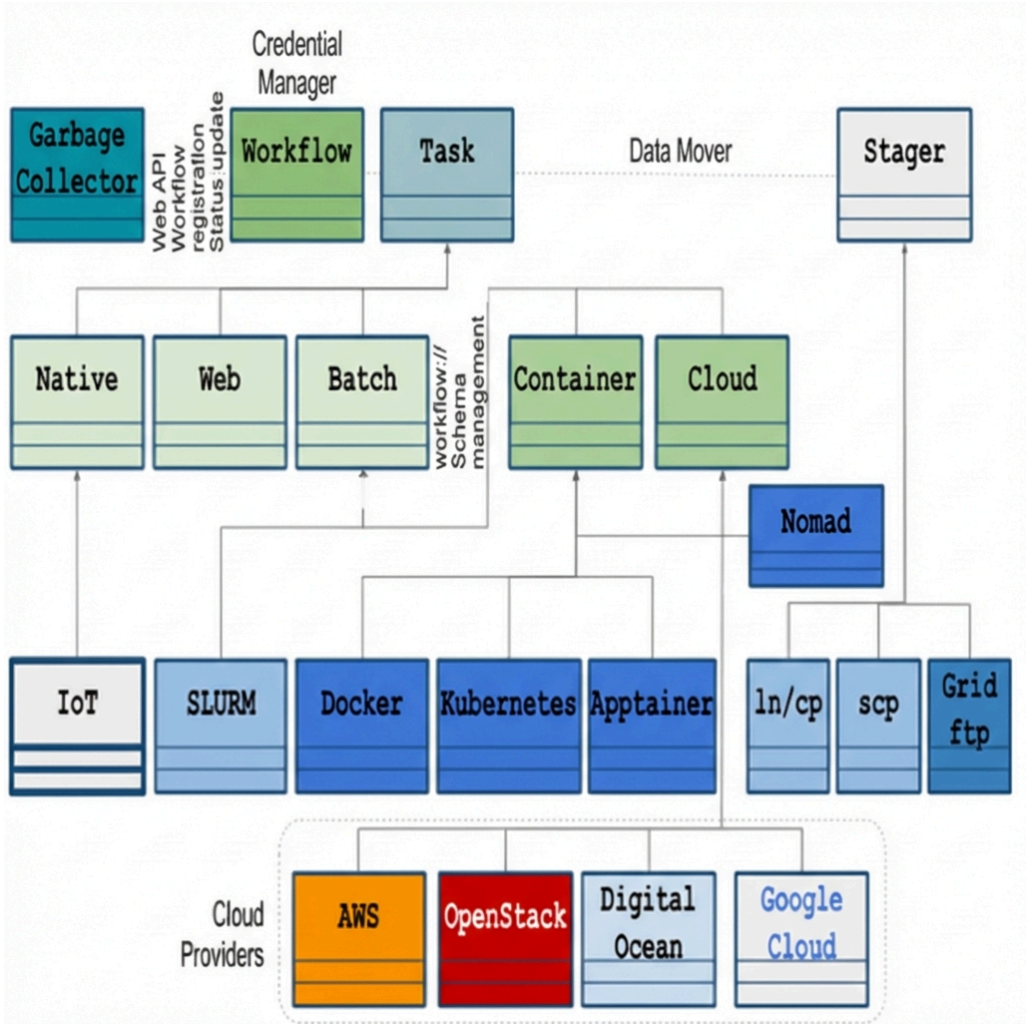


Figura 3.2
Diagrama de clases de las tareas desplegadas en contenedores en DAGonStar.

Esta jerarquía permite que el resto del motor trate de forma uniforme a todas las tareas, independientemente de si su ejecución se realiza de forma nativa, en una cola batch o dentro de un contenedor. El polimorfismo se utiliza para especializar únicamente

el comportamiento específico del *runtime*, reutilizando el resto de la infraestructura de planificación y gestión de dependencias.

3.3.2. Diseño de `ApptainerTask` y `RemoteApptainerTask`

La clase `ApptainerTask` se ha diseñado pensando en entornos HPC y Edge donde Apptainer [5] es el *runtime* preferente. Sus responsabilidades principales son:

- Verificar la disponibilidad de la imagen SIF requerida por la tarea, soportando la descarga y construcción dinámica desde URI de Docker (mediante `apptainer build`).
- Gestionar *overlays* de escritura mediante la creación de ficheros de imagen de tamaño configurable, permitiendo que aplicaciones con requisitos de escritura funcionen sobre imágenes SIF de solo lectura.
- Montar directorios de entrada y salida mediante áreas de transferencia temporal sincronizadas con montajes de tipo `bind`.
- Ejecutar el comando de usuario capturando códigos de salida y flujos de *log*, gestionando reintentos con privilegios (`sudo`) si la construcción inicial falla.

El Código 3.2 presenta la implementación detallada de la clase base local, incluyendo la lógica de preparación de entorno y ejecución de subprocesos.

```
import os, subprocess, uuid, threading, shutil
from dagon import Batch

class ApptainerTask(Batch):
    def __init__(self, name, command, image="docker://ubuntu:20.04", **
    kwargs):
        Task.__init__(self, name, command, **kwargs)
        self.image = image
        self.overlay_size = kwargs.get("overlay_size", "1024")
        self.container_id = f"{self.name}-{uuid.uuid4().hex[:8]}"
        self.work_dir = os.path.join("/tmp/dagon", self.container_id)
        self.sif_file = os.path.join(self.work_dir, f"{self.name}.sif")
        self.overlay_file = os.path.join(self.work_dir, "overlay.img")
        self._lock = threading.Lock()

    def _run_apptainer_command(self, cmd_args, capture_output=True):
        # Ejecuta comandos de Apptainer localmente
        return subprocess.run(cmd_args, capture_output=capture_output,
    text=True)
```

```
18
19      def create_container(self):
20          # Prepara el entorno (SIF + Overlay)
21          os.makedirs(self.work_dir, exist_ok=True)
22          self._prepare_sif_image()
23          self._create_overlay()
24
25      def _prepare_sif_image(self):
26          # Build dinámico si no existe
27          if not os.path.exists(self.sif_file):
28              self._run_apptainer_command(["apptainer", "build", self.
    sif_file, self.image])
29
30      def _create_overlay(self):
31          # Crea imagen de overlay para escritura
32          if not os.path.exists(self.overlay_file):
33              self._run_apptainer_command(["apptainer", "overlay", "
    create",
34                                           "--size", self.overlay_size,
    self.overlay_file])
35
36      def exec_in_container(self, command):
37          # Ejecución interna mediante 'apptainer exec'
38          exec_cmd = ["apptainer", "exec", "--overlay", self.overlay_file
    ,
39                      "--bind", f"{self.work_dir}:/work", self.sif_file,
    "bash", "-c", command]
40          return self._run_apptainer_command(exec_cmd).stdout
41
42      def export_file_to_staging(self, container_path, staging_filename):
43          # Extrae ficheros del contenedor sin usar montajes complejos
44          pass
45
46      def import_file_from_staging(self, staging_path, container_path):
47          # Introduce ficheros externos al entorno del contenedor
48          pass
49
50      def stage_in(self, src_task, src_path, dst_path):
51          # Copia entre contenedores (Data Mover)
52          pass
53
54      def cleanup_container(self):
55          # Eliminación de ficheros temporales SIF/Overlay
56          if os.path.exists(self.work_dir): shutil.rmtree(self.work_dir)
57
```

```
    def pre_process_command(self, command):
        # Intercepta workflow:/// antes de la ejecución
        return command

    def on_execute(self, script, script_name):
        self.create_container()
        return self.exec_in_container(self.command)

    def on_garbage(self):
        self.cleanup_container()
```

CÓDIGO 3.2. Implementación resumida de ApptainerTask para ejecución local

Por otro lado, la clase `RemoteApptainerTask` extiende el comportamiento anterior para el caso de ejecución distribuida en nodos remotos. Esta variante se encarga de:

- Gestionar el túnel SSH establecido por el motor para la orquestación remota del comando anterior.
- Delegar la ejecución de las utilidades de Apptainer en el nodo Edge, detectando la presencia del *runtime* en el sistema destino.
- Integrar el flujo de datos remoto, asegurando que los montajes *bind* apunten a los directorios del trabajador remoto y no del orquestador.

El Código 3.3 ilustra cómo se integra la lógica de Apptainer con la infraestructura de ejecución remota de DAGonStar.

```
from dagon.remote import RemoteTask

class RemoteApptainerTask(RemoteTask, ApptainerTask):
    def __init__(self, name, command, ip=None, ssh_username=None, **kwargs):
        RemoteTask.__init__(self, name=name, command=command, ip=ip, **kwargs)
        ApptainerTask.__init__(self, name, command, **kwargs)

    def _run_apptainer_command(self, cmd_args, capture_output=True):
        # Delega la ejecución vía SSH Connection
        cmd_str = " ".join(cmd_args)
        res = self.ssh_connection.execute_command(cmd_str)
        return type('Obj', (object,), {'stdout': res['output'], 'returncode': res['code']})

    def create_container(self):
```

```
        # Lógica remota idéntica pero sobre SSH
        super().create_container()

    def _prepare_sif_image(self):
        # Build remoto vía SSH
        super()._prepare_sif_image()

    def on_execute(self, script, script_name):
        # Orquestación remota del ciclo de vida
        RemoteTask.on_execute(self, script, script_name)
        return super().on_execute(script, script_name)

    def on_garbage(self):
        # Limpieza remota delegada
        super().on_garbage()
        RemoteTask.on_garbage(self)
```

CÓDIGO 3.3. Implementación de RemoteApptainerTask con todos sus métodos delegados

La diferencia fundamental reside en la redirección de las llamadas al ejecutable de Apptainer hacia el objeto `ssh_connection` establecido durante la fase de negociación de la tarea.

3.3.3. Diseño de `DockerTask` y `RemoteDockerTask`

La clase `DockerTask` proporciona soporte para la ejecución de tareas en contenedores Docker [6], aprovechando la versatilidad de este *runtime* en entornos Fog y Cloud. Sus funcionalidades clave incluyen:

- Interacción con el demonio de Docker mediante la librería oficial de Python[34], permitiendo la gestión programática de imágenes, contenedores y redes.
- Configuración de contenedores efímeros o persistentes, asignando volúmenes de forma dinámica para asegurar que el *working directory* de la tarea sea accesible desde el interior (montaje habitual en `/dagon`).
- Inyección de mecanismos de control: para evitar fallos por codificación de caracteres, se utiliza un script lanzador depositado en el contenedor mediante `docker cp`.
- Soporte para dispositivos hardware (GPU, sensores) mediante el parámetro `devices`, fundamental en tareas de inferencia en el Edge.

El Código 3.4 profundiza en la implementación técnica de la inyección de comandos y la gestión de volúmenes en el entorno local.

```python
import docker, os
from dagon import Batch

class DockerTask(Batch):
    def __init__(self, name, command, image=None, **kwargs):
        Task.__init__(self, name, command, **kwargs)
        self.client = docker.from_env()
        self.image = image or "ubuntu:22.04"
        self.container = None

    def include_command(self, body):
        # Inyección vía script temporal y 'docker exec'
        wrapper_path = os.path.join(self.working_dir, ".dagon_task.sh")
        with open(wrapper_path, "w") as f: f.write(body)
        return f"docker exec {self.container.id} bash /tmp/task.sh"

    def pre_process_command(self, command):
        # Lógica de creación del contenedor antes de la ejecución
        self.create_container()
        return command

    def pull_image(self, image):
        # Descarga de imagen desde el Registry
        return self.client.images.pull(image)

    def create_container(self):
        # Creación del contenedor persistente o efímero
        self.container = self.client.containers.run(
            self.image, detach=True, volumes={self.working_dir: {'bind'
    : '/dagon'}})

    def get_running_container(self):
        # Recupera objeto contenedor si ya existe
        return self.client.containers.get(self.container.id)

    def remove_container(self):
        # Borrado forzoso
        if self.container: self.container.remove(force=True)

    def on_execute(self, script, script_name):
        # Flujo de ejecución principal
        return {'code': 0, 'output': "OK"}

    def on_garbage(self):
```

```
            self.remove_container()
```

CÓDIGO 3.4. Implementación resumida de DockerTask para entornos locales

Por su parte, `RemoteDockerTask` extiende esta lógica para orquestar contenedores en nodos remotos. Aprovechando que el SDK de Docker soporta transporte SSH, esta clase se encarga de:

- Instanciar el cliente remoto utilizando una URL con esquema `ssh://`, lo que permite que todas las llamadas a la API (crear, listar, borrar) se tunelen de forma transparente.
- Gestionar la transferencia de scripts hacia la máquina Edge antes de la ejecución del contenedor.
- Centralizar los logs remotos, redirigiendo la salida estándar del demonio Docker remoto hacia el panel de monitorización del orquestador.

El Código 3.5 presenta el diseño de la variante remota y su integración con el sistema de transporte seguro de DAGonStar.

```python
import docker
from dagon.remote import RemoteTask

class DockerRemoteTask(RemoteTask, DockerTask):
    def __init__(self, name, command, ip=None, ssh_username=None, **kwargs):
        DockerTask.__init__(self, name, command, **kwargs)
        RemoteTask.__init__(self, name=name, ip=ip, **kwargs)
        # Cliente conectado vía túnel SSH
        self.client = docker.DockerClient(base_url=f"ssh://{ssh_username}@{ip}")

    def on_execute(self, launcher_script, script_name):
        # Orquestación remota del script y contenedor
        RemoteTask.on_execute(self, launcher_script, script_name)
        return DockerTask.on_execute(self, launcher_script, script_name)

    def on_garbage(self):
        # Limpieza binaria: remota (SSH) + contenedor (SDK)
        DockerTask.on_garbage(self)
        RemoteTask.on_garbage(self)
```

CÓDIGO 3.5. Implementación de DockerRemoteTask con soporte remoto resumido

La arquitectura permite que la lógica de orquestación sea agnóstica a la ubicación del demonio, garantizando que el usuario defina la tarea del mismo modo tanto en local como en remoto.

3.3.4. Diseño de `KubernetesTask` y `RemoteKubernetesTask`

Para la integración con Kubernetes [8] se ha optado por un patrón de *pod ejecutor persistente*, lo que mejora significativamente los tiempos de respuesta al evitar el proceso de *scheduling* y descarga de imágenes en cada tarea. Las funciones de `KubernetesTask` son:

- Orquestar la creación de un *pod* con una configuración de espera infinita (`sleep infinity`) utilizando la SDK oficial[35].
- Implementar el acceso a recursos del host mediante *hostPath* para facilitar el *staging* de datos.
- Gestionar la ejecución de comandos a través de la API de *streaming*, permitiendo una interacción síncrona con el proceso dentro del contenedor.
- Monitorear el estado de salud del pod y garantizar su eliminación tras el flujo de trabajo.

El Código 3.6 resume la lógica de interacción local con el clúster (típicamente a través de un archivo *kubeconfig* local).

```python
import uuid, time
from kubernetes import client, config
from kubernetes.stream import stream
from dagon import Batch

class KubernetesTask(Batch):
    def __init__(self, name, command, image="ubuntu:22.04", **kwargs):
        Task.__init__(self, name, command, **kwargs)
        config.load_kube_config()
        self.api = client.CoreV1Api()
        self.pod_name = f"dagtask-{uuid.uuid4().hex[:6]}"

    def create_pod(self):
        # Creación y espera de pod en estado Running
        pass

    def exec_in_pod(self, command):
        # Streaming API para interactuar con el contenedor
```

```
19          return stream(self.api.connect_get_namespaced_pod_exec, self.
    pod_name, "default",
20                        command=["/bin/bash", "-c", command], stderr=True
    , stdout=True)
21
22      def stage_in(self, src_task, src_path, dst_path):
23          # Movimiento de datos entre pods (cat/tar pipe)
24          pass
25
26      def remove_pod(self):
27          # Eliminación del recurso en el namespace
28          self.api.delete_namespaced_pod(name=self.pod_name, namespace="
    default")
29
30      def pre_process_command(self, command):
31          return command
32
33      def on_execute(self, script, script_name):
34          self.create_pod()
35          return self.exec_in_pod(self.command)
36
37      def on_garbage(self):
38          self.remove_pod()
```

CÓDIGO 3.6. Implementación resumida de KubernetesTask para entornos locales

La variante `RemoteKubernetesTask` aborda la problemática de clústeres Edge (como k3s) situados en redes externas. En lugar de requerir que el orquestador tenga acceso directo a la API de Kubernetes remota, esta clase utiliza un patrón de delegación por SSH:

- Emplea comandos de la CLI `kubectl` instalada en el nodo maestro remoto mediante SSH.
- No requiere la exposición del puerto de la API de Kubernetes hacia el exterior, mejorando la seguridad del clúster Edge.
- Facilita la orquestación en entornos donde el orquestador y el clúster están separados por cortafuegos o NAT.

El Código 3.7 muestra cómo se especializa la gestión de comandos para este escenario distribuido.

```
1  from dagon.remote import RemoteTask
2
3  class RemoteKubernetesTask(RemoteTask, KubernetesTask):
```

```
    def __init__(self, name, command, ip=None, **kwargs):
        RemoteTask.__init__(self, name=name, command=command, ip=ip, **kwargs)
        self.pod_name = f"remotepod-{self.name.lower()}"

    def _run_kubectl_command(self, cmd_args):
        # Orquestación vía SSH
        return self.ssh_connection.execute_command("kubectl " + " ".join(cmd_args))

    def create_pod(self):
        # Lanzamiento remoto vía kubectl run
        self._run_kubectl_command(["run", self.pod_name, "--image=" + self.image])

    def exec_in_pod(self, command):
        # Invocación de kubectl exec remoto
        return self._run_kubectl_command(["exec", self.pod_name, "--", "bash", "-c", command])

    def stage_in(self, src_task, src_path, dst_path):
        # Streaming remoto entre nodos Edge
        pass

    def remove_pod(self):
        self._run_kubectl_command(["delete", "pod", self.pod_name])

    def pre_process_command(self, command):
        return command

    def on_execute(self, script, script_name):
        RemoteTask.on_execute(self, script, script_name)
        return super().on_execute(script, script_name)

    def on_garbage(self):
        self.remove_pod()
        RemoteTask.on_garbage(self)
```

CÓDIGO 3.7. Implementación de RemoteKubernetesTask con soporte administrativo remoto

3.3.5. Diseño de `NomadTask` y `RemoteNomadTask`

La clase `NomadTask` se integra con HashiCorp Nomad [7] para la ejecución desacoplada en clústeres Edge de recursos limitados. A diferencia de Kubernetes, Nomad utiliza

un enfoque centrado en *Jobs* que se envían a través de una API REST. Sus funciones principales incluyen:

- Generar la especificación dinámica del trabajo en formato JSON [36], definiendo el driver deseado (típicamente `docker` o `exec`) y los artefactos necesarios.
- Realizar el seguimiento de las *allocations* (instancias del trabajo) para capturar el estado final de la ejecución.
- Recuperar los flujos de logs mediante peticiones HTTP asíncronas hacia el agente Nomad, empleando la librería *Requests*[37].
- Gestionar la prioridad y los centros de datos definidos en el clúster.

El Código 3.8 detalla el diseño de la sumisión de trabajos y el polling de estado sobre un agente local.

```python
import requests, json, time, uuid
from dagon import Batch

class NomadTask(Batch):
    def __init__(self, name, command, image="ubuntu:20.04", nomad_addr="http://localhost:4646"):
        Task.__init__(self, name, command)
        self.nomad_address = nomad_addr.rstrip('/')
        self.job_id = f"dag-job-{uuid.uuid4().hex[:8]}"

    def _generate_job_spec(self):
        # Especificación HCL/JSON del Job
        pass

    def _submit_job(self):
        # Invocación de /v1/jobs vía POST
        return requests.post(f"{self.nomad_address}/v1/jobs", json=self._generate_job_spec())

    def _wait_for_completion(self, timeout=3600):
        # Polling de ClientStatus de las allocations
        pass

    def _get_logs(self):
        # Captura de stdout/stderr vía API
        pass

    def _cleanup_job(self):
```

```
        # Purga el registro del Job en Nomad
        requests.delete(f"{self.nomad_address}/v1/job/{self.job_id}?
    purge=true")

    def include_command(self, body):
        return body

    def pre_process_command(self, command):
        return command

    def on_execute(self, script, script_name):
        self._submit_job()
        self._wait_for_completion()
        return {'output': self._get_logs(), 'code': 0}

    def on_garbage(self):
        self._cleanup_job()
```

CÓDIGO 3.8. Implementación técnica resumida de NomadTask local

La clase `RemoteNomadTask` extiende esta funcionalidad para casos donde el agente Nomad se encuentra en un nodo remoto. La estrategia de diseño se apoya en la naturaleza distribuida de Nomad:

- Construye la dirección de la API de forma dinámica utilizando la IP del nodo Edge objetivo.
- Utiliza el protocolo HTTP sobre la red gestionada por el orquestador, manteniendo la misma lógica de sumisión que la versión local.
- Permite una transición transparente entre ejecución local y remota sin modificar la definición del *workflow*.

El Código 3.9 muestra la inicialización y el flujo de ejecución para la variante remota.

```
from dagon.remote import RemoteTask

class RemoteNomadTask(RemoteTask, NomadTask):
    def __init__(self, name, command, ip=None, **kwargs):
        # IP dinámica para el endpoint de la API
        nomad_address = f"http://{ip}:4646"
        NomadTask.__init__(self, name, command, nomad_address=
    nomad_address, **kwargs)
        RemoteTask.__init__(self, name=name, ip=ip, **kwargs)

```

```
    def on_execute(self, script, script_name):
        # Orquestación SSH + Sumisión HTTP
        RemoteTask.on_execute(self, script, script_name)
        return NomadTask.on_execute(self, script, script_name)

    def on_garbage(self):
        # Limpieza coordinada
        NomadTask.on_garbage(self)
        RemoteTask.on_garbage(self)
```

CÓDIGO 3.9. Implementación de RemoteNomadTask para agentes remotos

Esta uniformidad en el diseño demuestra la potencia de la abstracción de tareas en DAGonStar, permitiendo orquestar gestores complejos con un impacto mínimo en el código del orquestador.

3.3.6. Validación funcional inicial: Scripts de Taskflow y Dataflow

Como complemento al diseño de las clases de tareas, se han implementado una serie de *scripts* auxiliares localizados en los directorios `examples/taskflow` y `examples/dataflow`. Estos componentes, aunque no forman parte de la evaluación de rendimiento detallada en el próximo capítulo, han sido esenciales para validar la funcionalidad básica del motor y el movimiento de datos en condiciones de uso habituales de un PC personal.

Los scripts de **`taskflow`** se orientan a la validación de la orquestación básica y el soporte multitecnología. Siguiendo un patrón común, definen un objeto `Workflow`, instancian tareas de tipo **`DagonTask`** especificando el *runtime* deseado (`DOCKER`, `APPTAINER`, `K3S`) y establecen las dependencias de forma explícita mediante el método `add_dependency_to`. El objetivo de estas pruebas es verificar que el motor es capaz de lanzar comandos secuencialmente (como `/bin/hostname` o `uptime`) dentro de los contenedores y gestionar correctamente los códigos de salida. El Código 3.10 muestra un ejemplo simplificado de orquestación sobre un clúster k3s local.

```
import json
from dagon import Workflow
from dagon.kubernetes_task import KubernetesTask

if __name__ == '__main__':
    workflow = Workflow("Taskflow-Demo-K3s")

    taskA = KubernetesTask("Tokyo", "/bin/hostname")
    taskB = KubernetesTask("Berlin", "/bin/date")
    taskC = KubernetesTask("Nairobi", "/usr/bin/uptime")
    taskD = KubernetesTask("Moscow", "/bin/uname -a")

```

```
13      workflow.add_task(taskA)
14      workflow.add_task(taskB)
15      workflow.add_task(taskC)
16      workflow.add_task(taskD)
17
18      # Definición de dependencias explícitas
19      taskB.add_dependency_to(taskA)
20      taskC.add_dependency_to(taskA)
21      taskD.add_dependency_to(taskB)
22      taskD.add_dependency_to(taskC)
23
24      # Exportación del workflow a JSON y ejecución
25      jsonWorkflow = workflow.as_json()
26      with open('taskflow-demo-k3s.json', 'w') as outfile:
27          stringWorkflow = json.dumps(jsonWorkflow, sort_keys=True,
        indent=2)
28          outfile.write(stringWorkflow)
29
30      workflow.run()
```

CÓDIGO 3.10. Script resumido de orquestación con Taskflow sobre k3s

Por su parte, los scripts de **dataflow** profundizan en la gestión del flujo de datos mediante el esquema de URI (Identificador de Recursos Uniforme) **workflow:///**. El esquema **workflow:///** actúa como un espacio de nombres global dentro del flujo de trabajo, permitiendo que cualquier tarea referencie ficheros generados por sus predecesoras sin conocer de antemano la ruta absoluta en el host o en el contenedor. Cuando el motor invoca el método `make_dependencies()`, realiza un análisis sintáctico de los comandos buscando este patrón; si detecta una referencia a una tarea `A`, establece automáticamente la precedencia. Durante la fase de ejecución, el *Stager* intercepta estas URIs y las resuelve dinámicamente: si los datos están en un nodo remoto, activa el protocolo de transferencia necesario (`scp` o *streaming*) para depositar una copia en el directorio de trabajo de la tarea actual antes de lanzar el contenedor. A diferencia de los anteriores, estos scripts suelen utilizar el método `make_dependencies()`, que infiere automáticamente la jerarquía de ejecución analizando las referencias a ficheros de otras tareas en los comandos. P. ej., una tarea puede generar un fichero (`hostname >f1.txt`) que es consumido por la siguiente mediante una referencia relativa (`cat workflow:///A/f1.txt`). Estas pruebas validan de forma crítica el subsistema de *staging* local y remoto, incluyendo casos avanzados como el uso de *checkpoints* persistentes en entornos Docker y la verificación post-ejecución del contenido de los directorios de trabajo (*scratch directories*). Estas pruebas garantizan que la abstracción de tareas desplegadas en contenedores es interoperable con el sistema de movimiento de ficheros original de DAGonStar. El Código 3.11 ilustra la gestión de dependencias implícitas en un entorno Kubernetes remoto.

```python
import json
import configparser
from dagon import Workflow
from dagon.task import DagonTask, TaskType

if __name__ == '__main__':
    # Configuración de conexión remota
    config = configparser.ConfigParser()
    config.read('dagon.ini')
    REMOTE_IP = config.get('ssh', 'remote_ip')
    SSH_USER = config.get('ssh', 'ssh_user')
    SSH_PORT = config.getint('ssh', 'ssh_port')

    workflow = Workflow("DataFlow-Demo-K0S-Remote")

    # Tarea A: Genera un fichero de salida en nodo remoto
    taskA = DagonTask(TaskType.KUBERNETES, "A",
                      "mkdir output; hostname > output/f1.txt",
                      image="ubuntu:20.04", ip=REMOTE_IP,
                      ssh_username=SSH_USER, ssh_port=SSH_PORT)

    # Tarea B: Consume el fichero de A mediante workflow:///
    taskB = DagonTask(TaskType.KUBERNETES, "B",
                      "echo $RANDOM > f2.txt; cat workflow:///A/output/
    f1.txt >> f2.txt",
                      image="ubuntu:20.04", ip=REMOTE_IP,
                      ssh_username=SSH_USER, ssh_port=SSH_PORT)

    # Tarea C: Tarea paralela que también consume de A
    taskC = DagonTask(TaskType.KUBERNETES, "C",
                      "echo $RANDOM > f2.txt; cat workflow:///A/output/
    f1.txt >> f2.txt",
                      image="ubuntu:20.04", ip=REMOTE_IP,
                      ssh_username=SSH_USER, ssh_port=SSH_PORT)

    workflow.add_task(taskA)
    workflow.add_task(taskB)
    workflow.add_task(taskC)

    # Inferencia automática de dependencias y ejecución
    workflow.make_dependencies()
    workflow.run()
```

CÓDIGO 3.11. Script resumido de Dataflow con dependencias implícitas sobre k0s remoto

3.4. Gestión de Credenciales y Parámetros: `dagon.ini`

El despliegue de tareas en recursos heterogéneos (Edge, Fog, Cloud) requiere gestionar múltiples tipos de credenciales y parámetros de red: claves SSH para nodos remotos, *tokens* de Grafana, cadenas de conexión a MongoDB Atlas [16] o direcciones de los orquestadores Nomad y Kubernetes.

Para centralizar esta información y evitar que datos sensibles se incluyan directamente en el código de los *workflows*, DAGonStar utiliza un archivo de configuración centralizado denominado `dagon.ini`. Este archivo sigue el formato estándar INI de Python [18], organizado en secciones que las diferentes clases de tareas consultan dinámicamente durante su inicialización.

El Código 3.12 ilustra la estructura de este archivo, mostrando cómo se definen los accesos a los distintos componentes del ecosistema de forma desacoplada.

```
[ssh]
# Credenciales para nodos Edge (Raspberry Pi)
raspi_ip = 192.168.1.XX
raspi_user = valid_user
raspi_port = 22

[mongodb]
# Conexion a base de datos de metricas (Atlas)
uri = mongodb+srv://db_user:db_password@cluster0.example.net/
database = DagOn_Metrics
collection = workflow_results

[grafana]
# Integracion para visualizacion de dashboards
url = http://localhost:3000
api_token = glsa_xxxx_DUMMY_TOKEN_xxxx
dashboard_uid = energy-analysis-dashboard

[prometheus]
# Servidor de series temporales para monitorizacion energetica
url = http://localhost:9090
sampling_interval = 2

[nomad]
# Direcciones de los agentes Nomad en las distintas capas
edge_address = http://192.168.1.XX:4646
fog_address = http://localhost:4646
datacenter = dc1

```

```
[batch]
# Configuracion de directorios de trabajo locales
scratch_dir_base = /tmp/dagon_scratch/
remove_dir = True
```

CÓDIGO 3.12. Ejemplo del archivo de configuración `dagon.ini` con valores ficticios

Esta aproximación permite que los ingenieros de datos y desarrolladores de *workflows* puedan intercambiar el entorno de ejecución (p. ej., pasar de una Raspberry Pi a un clúster de servidores) simplemente modificando el archivo `dagon.ini`, sin necesidad de alterar una sola línea de código en la definición de la lógica de negocio del flujo de trabajo.

3.5. Diseño del Mecanismo de *Staging* Transversal

3.5.1. Data Mover y operaciones básicas

La arquitectura distingue dos niveles de responsabilidad en la gestión del movimiento de datos: el *Stager* y el *Data Mover*. El ***Stager*** es el componente de orquestación de alto nivel que, basándose en el grafo de dependencias y la resolución de las URIs `workflow:///`, decide qué ficheros deben transferirse y en qué momento del ciclo de vida de la tarea. Por su parte, el **Data Mover** es el componente de bajo nivel encargado de la ejecución técnica de dichas transferencias; implementa una interfaz abstracta para las operaciones de entrada/salida, abstrayendo al resto del sistema del protocolo de transporte utilizado (`scp`, copia local, etc.). El componente **Data Mover** implementa una interfaz abstracta para las operaciones de entrada/salida de datos asociadas a las tareas. Sus primitivas básicas son:

- `stage_in(src, dst)`: garantiza que los ficheros requeridos por una tarea estén disponibles en el entorno de ejecución antes de lanzarla.
- `stage_out(src, dst)`: recoge los resultados generados por una tarea y los pone a disposición de las tareas dependientes.
- `cleanup(path)`: elimina ficheros o directorios temporales una vez que ya no son necesarios.

Estas operaciones pueden tener múltiples implementaciones dependiendo del origen y destino de los datos: copias locales (`ln`/`cp`), transferencia segura mediante `scp`, sincronización con `rsync` o *streaming* directo entre contenedores. La Figura 3.3 muestra la relación entre el *Stager*, el *Data Mover* y los distintos backends de transferencia.

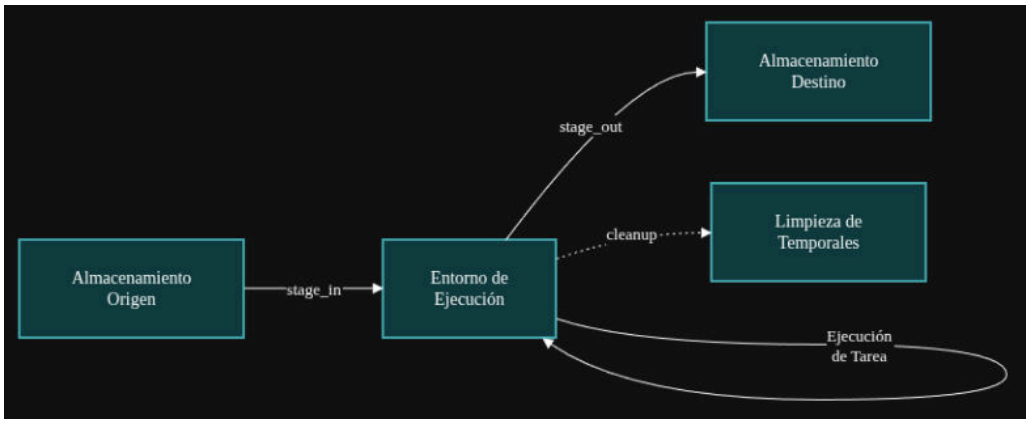


Figura 3.3
Esquema del componente Stager *y sus operaciones de* staging.

3.5.2. Staging para Apptainer, Docker y Nomad

En el caso de Apptainer, Docker y Nomad, la solución se basa en directorios de intercambio ubicados en el sistema de archivos del host. Cada tarea dispone de un directorio *staging* local, que se monta en el contenedor como volumen de entrada/salida. El flujo típico es el siguiente:

1. El *Stager* copia o enlaza los ficheros producidos por las tareas predecesoras en el directorio *staging* de la tarea actual.
2. La tarea se ejecuta en el contenedor, accediendo a los ficheros a través del volumen montado.
3. Al finalizar, los resultados escritos en el volumen se recogen desde el host y se registran como salidas de la tarea.

Este patrón permite que contenedores ejecutados en nodos distintos se comuniquen a través de copias entre directorios del host, sin requerir un sistema de archivos distribuido global.

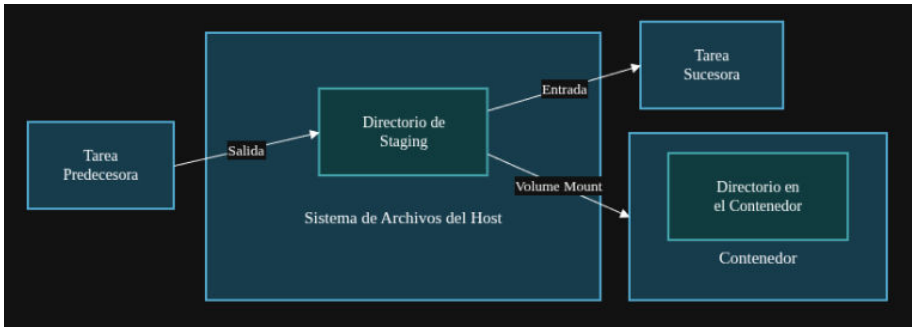


Figura 3.4
Mecanismo de staging *basado en volúmenes para Docker, Apptainer y Nomad.*

3.5.3. Staging para Kubernetes: *Streaming* entre pods

En el entorno Kubernetes se ha optado por un mecanismo de *staging* basado en *streaming* entre pods, especialmente efectivo para ficheros de tamaño pequeño o medio. El procedimiento general es:

1. El *Stager* solicita la descarga de un fichero desde un pod origen utilizando `kubectl exec` combinado con comandos `cat` o `tar`.
2. El flujo de bytes se redirige a la máquina del orquestador, donde se puede almacenar temporalmente o encadenar directamente hacia un pod destino.
3. En el pod destino se reconstruye el fichero mediante otro comando `cat` o `tar -x`, sin necesidad de utilizar un volumen compartido.

Este enfoque reduce la dependencia de sistemas de almacenamiento compartido como NFS [38] o Ceph [39] y evita cuellos de botella asociados a volúmenes de uso general. No obstante, la arquitectura permite también configurar backends alternativos basados en volúmenes persistentes para escenarios con ficheros de mayor tamaño.

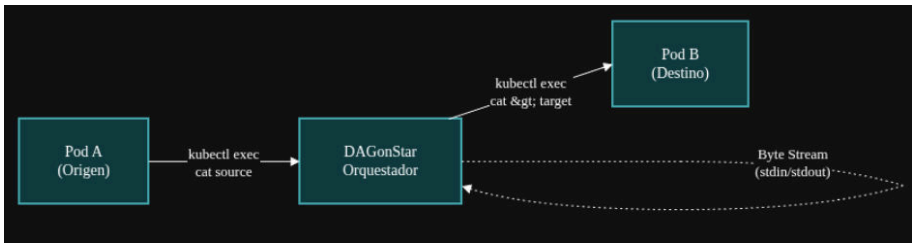


Figura 3.5
Mecanismo de staging *basado en* streaming *entre pods para Kubernetes.*

3.6. Módulo de Monitorización y Trazabilidad

El diseño incluye un módulo de monitorización integrado basado en Prometheus y Grafana. Cada tarea expone métricas como tiempo de inicio, tiempo de finalización, estado (éxito o fallo) y volumen de datos procesados. En el caso de la Raspberry Pi, se han añadido métricas de consumo energético obtenidas a partir del controlador de gestión de potencia del dispositivo.

Estas métricas se recogen periódicamente mediante *exporters* —agentes que extraen métricas del sistema y las exponen en un formato compatible con Prometheus— y se almacenan en su base de datos de series temporales [11]. Grafana [12] se utiliza para construir paneles que muestran, p. ej., el consumo energético por tipo de contenedor, la latencia de *staging* entre distintas capas del ecosistema *edge-to-cloud* o la distribución de tiempos de ejecución por tarea. Esta información no solo es útil para la evaluación

experimental, sino que sienta las bases para futuros mecanismos de planificación *energy-aware* [4].

3.7. Síntesis de las Decisiones de Diseño

El diseño de la solución puede resumirse en las siguientes decisiones clave:

- Mantener el modelo de programación original de DAGonStar, introduciendo nuevas clases de tareas que encapsulan la lógica específica de contenedores sin modificar la interfaz que percibe el usuario.
- Introducir un componente de *Data Mover* y un *Stager* transversales, capaces de combinar diferentes técnicas de transferencia de datos según el entorno de ejecución (copia local, SCP, *streaming* entre pods).
- Centralizar la gestión de credenciales y parámetros de entorno en el archivo `dagon.ini`, desacoplando la seguridad y la infraestructura de la lógica de las tareas.
- Integrar de forma nativa la monitorización de rendimiento y energía, de modo que cada ejecución de un *workflow* genere automáticamente los datos necesarios para su análisis posterior.

Estas decisiones constituyen la base sobre la que se apoya la implementación descrita en el capítulo siguiente y orientan la evaluación experimental que se presenta más adelante.

4. EVALUACIÓN

En este capítulo se presenta la evaluación de la solución propuesta. En primer lugar, se describe la metodología general seguida para validar el funcionamiento del sistema y recoger las métricas de interés. A continuación, se analiza el comportamiento funcional del *workflow* Fog–Edge–Cloud y, finalmente, se estudia el rendimiento de los distintos componentes, con especial atención al impacto de las tecnologías de contenedores virtuales y de los mecanismos de *staging*.

4.1. Metodología de Evaluación

La evaluación se ha planteado con un enfoque mixto, cuantitativo y cualitativo, similar al adoptado en otros Trabajos Fin de Grado de la misma titulación [40], [41]. El objetivo es comprobar que la arquitectura diseñada cumple los requisitos funcionales y, al mismo tiempo, cuantificar el coste que introduce en términos de tiempo de ejecución y uso de recursos.

4.1.1. Escenario experimental

El escenario de prueba se basa en un *workflow* dividido en tres tareas principales:

- **Tarea A (Edge – Contenedor/Orquestador):** captura periódica de datos de un sensor de temperatura y humedad conectado a una Raspberry Pi.
- **Tarea B (Edge – Contenedor/Orquestador):** preprocesamiento local de los datos, aplicando filtrado, agregación y generación de métricas intermedias.
- **Tarea C (Fog/Cloud – Contenedor):** persistencia de los resultados en una base de datos MongoDB Atlas [16] y exposición de métricas para Prometheus.



Figura 4.1

Montaje físico del nodo Edge: Raspberry Pi 5 conectada a Arduino con sensor DHT11 [42].

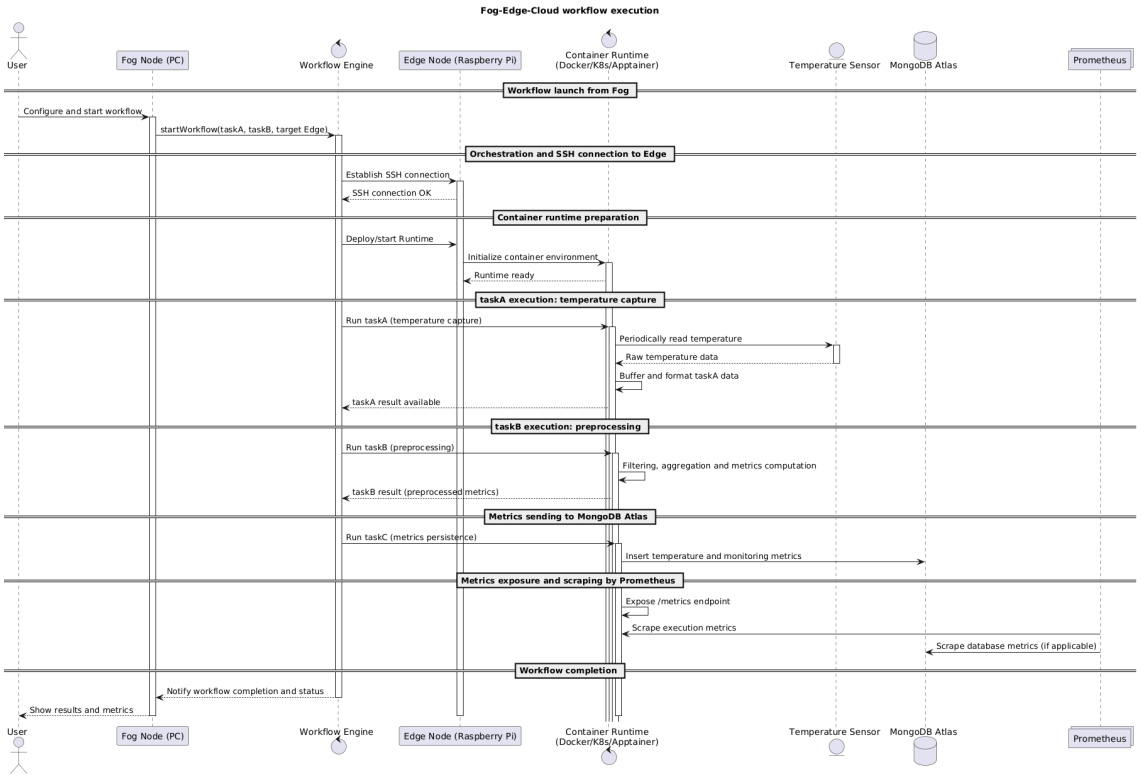


Figura 4.2

Diagrama de secuencia del workflow Fog–Edge–Cloud.

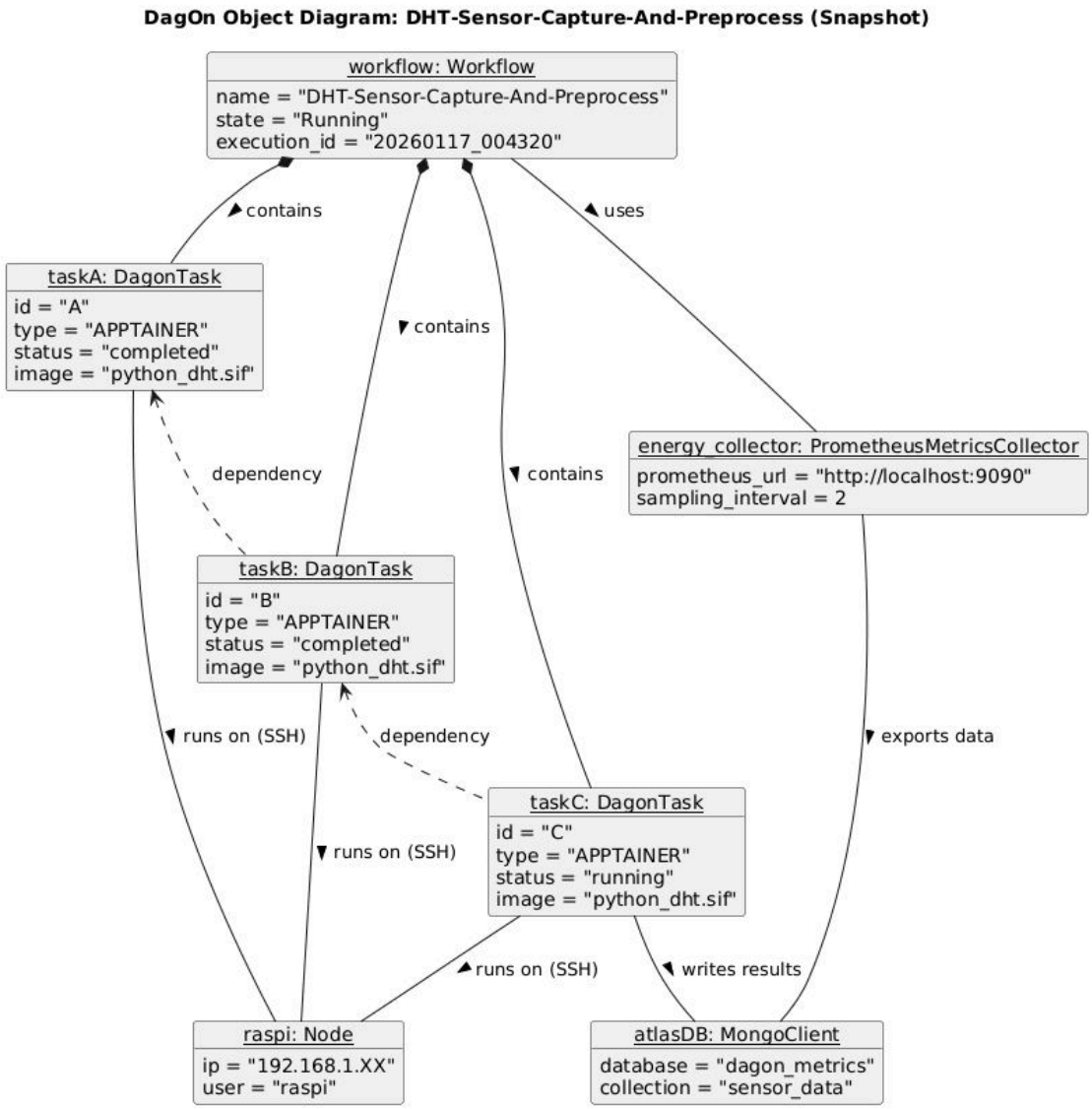


Figura 4.3
Diagrama de objetos del workflow.

Las tres tareas se encapsulan como instancias de `ApptainerTask`, `DockerTask`, `KubernetesTask` o `NomadTask`, según la configuración, y se conectan mediante dependencias de datos gestionadas por el *Stager*. En la Figura 4.1 se muestra el detalle del hardware real utilizado para el nodo Edge, mientras que las Figuras 4.2 y 4.3 ilustran el flujo completo de ejecución desde el punto de vista del motor de *workflow*.

4.1.2. Infraestructura hardware y software

Para la realización de las pruebas se ha empleado un nodo Edge basado en una **Raspberry Pi 5 Model B (Rev 1.1)** [13]. Las especificaciones técnicas y la configuración del entorno se resumen a continuación:

- **Sistema Operativo**: Debian GNU/Linux 13 (trixie) [43], proporcionando un en-

torno actualizado para el soporte de las últimas versiones de los *runtimes* de contenedores.

- **Almacenamiento**: se utiliza una tarjeta MicroSD de 8 GB (Fabricante ID: `0x000002`, Modelo: `SA08G`), con un particionado que reserva 512 MB para el *firmware* de arranque y 6.7 GB para el sistema de archivos raíz (`/`).
- **Gestión de Memoria**: se ha configurado un dispositivo `zram` de 2 GB para actuar como área de intercambio (*swap*) comprimida en RAM, optimizando el rendimiento ante picos de carga de trabajo.
- **Hardware Periférico**: se ha integrado un microcontrolador **Arduino Uno** [44] encargado de la lectura del sensor **DHT11** [42]. La comunicación con el nodo Edge se realiza mediante un enlace serie vía `/dev/ttyACM0` a una velocidad de 9600 baudios.

El código cargado en el microcontrolador (Código 4.1) actúa como un puente de telemetría de bajo nivel. Implementado en el lenguaje C++ [45] sobre la plataforma Arduino [46], su función principal es realizar el muestreo periódico del sensor DHT11 utilizando la librería oficial de Adafruit [47], gestionar los posibles errores de lectura (*checksum errors* o *timeouts*) y formatear los datos en una cadena de texto estructurada que será posteriormente parseada por la Tarea A en el nodo Raspberry Pi.

```
#include "DHT.h"

#define DHTPIN 8        // Pin digital conectado al DHT11
#define DHTTYPE DHT11   // Tipo de sensor

DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  Serial.println("Starting DHT11 sensor reading...");
  dht.begin();
}

void loop() {
  delay(1000); // Espera 1 segundo entre lecturas

  float h = dht.readHumidity();
  float t = dht.readTemperature(); // Temperatura en Celsius

  if (isnan(h) || isnan(t)) {
    Serial.println("Error reading from DHT11 sensor");
    return;
```

```
    }

    // Formato de salida para parseo en el Edge
    Serial.print("Humidity: ");
    Serial.print(h);
    Serial.print(" %  |  Temperature: ");
    Serial.print(t);
    Serial.println(" C");
}
```

CÓDIGO 4.1. Sketch de Arduino para la captura y envío serie de datos del DHT11

4.1.3. Configuración y despliegue de los entornos de ejecución

La puesta a punto de los cuatro entornos de ejecución sobre la arquitectura ARM64 de la Raspberry Pi 5 presentó distintos niveles de complejidad y carga de trabajo para el dispositivo:

- **Docker**: representó el despliegue de menor complejidad, realizado mediante el script oficial de conveniencia (`curl -sSL https://get.docker.com | sh`). El soporte nativo para ARM64 y la madurez de la herramienta permitieron una configuración inmediata sin ajustes adicionales de kernel. La única configuración post-instalación fue la adición del usuario al grupo `docker` para evitar el uso sistemático de privilegios elevados en las tareas de DAGonStar [6].

- **Apptainer (v1.4.4)**: constituyó el proceso de mayor complejidad técnica y carga computacional. Aunque Apptainer es el estándar de facto en entornos HPC, su despliegue en arquitecturas ARM64 para Edge Computing aún enfrenta retos de empaquetado.

  Debido a la ausencia de binarios oficiales para Debian 13 (Trixie), se realizó un despliegue siguiendo el flujo de compilación del repositorio oficial en GitHub ([48]). Este proceso requirió la instalación de Go [49] (v1.22+) y un conjunto extenso de dependencias de desarrollo (*libseccomp-dev [50], squashfs-tools [51], fuse3 [52], pkg-config [53]*). La secuencia de comandos crítica involucró la configuración del proyecto y la compilación paralela:

```
# Configuracion del binario (prefijo /usr/local)
./mconfig --prefix=/usr/local

# Compilacion distribuida en los 4 nucleos (RPi 5)
cd builddir && make -j4
sudo make install

```

```
# Gestion de dependencias FUSE \cite{fuse_docs} para montajes sin
    root
cd ..
./scripts/download-dependencies
./scripts/compile-dependencies
sudo ./scripts/install-dependencies

```

Esta instalación permite que Apptainer opere en modo *rootless* (sin privilegios), aprovechando las capacidades de *user namespaces* del kernel de Debian 13. Es fundamental para la seguridad en el Edge, ya que evita que un contenedor comprometido obtenga acceso root al sistema host. Además, se configuró el sistema para trabajar con el formato `.sif` (*Singularity Image Format*), que ofrece mayor integridad y rendimiento en sistemas de archivos locales comparado con las capas de Docker. Esta elección tecnológica garantiza el aislamiento de los procesos de captura de sensores DHT11 frente al sistema operativo base.

La compilación empleó los 4 núcleos del dispositivo al máximo de su capacidad durante aproximadamente 45 minutos, lo que evidenció el impacto térmico inicial de la puesta a punto. Tras la instalación, se ejecutó un protocolo de limpieza (Código 4.2) para liberar espacio en la tarjeta SD [5].

```
# 1. Remove Go build caches
sudo rm -rf /root/.cache/go-build
sudo rm -rf ~/.cache/go-build

# 2. Delete the Go workspace directory
sudo rm -rf ~/go

# 3. Clean up k3s remnants (if previously uninstalled)
sudo rm -rf /var/log/pods
sudo rm -rf /var/log/containers

# 4. APT package manager maintenance
sudo apt clean
sudo apt autoremove -y

```

CÓDIGO 4.2. Protocolo de limpieza postinstalación en RPi 5

- **Kubernetes (k3s)**: se seleccionó la distribución `k3s` ([54]) por su baja huella de memoria. Para optimizar el rendimiento en la Raspberry Pi, se utilizó el motor de base de datos embebido **SQLite** [55] en lugar de etcd, reduciendo drásticamente el uso de CPU e I/O de disco. Asimismo, se habilitó el *Local Path Provisioner* para la

gestión de volúmenes persistentes y se configuró el plugin de red **Flannel** en modo `host-gw` para minimizar la latencia de red entre pods.

Un paso esencial en el Edge fue la habilitación de los *cgroups* en el arranque. Sin los parámetros `cgroup_memory=1` y `cgroup_enable=memory` en el fichero `/boot/firmware/cmdline.txt`, el binario de Kubernetes no puede iniciar el *runtime* de contenedores adecuadamente. Además, para facilitar la integración con DAGonStar sin requerir `sudo` en cada llamada, se configuró el acceso local al cluster:

```
# Configuracion del cliente kubectl para el usuario actual
mkdir -p ~/.kube
sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/config
sudo chown $(id -u):$(id -g) ~/.kube/config
export KUBECONFIG=$HOME/.kube/config
```

CÓDIGO 4.3. Configuración de acceso local a k3s

- **Kubernetes (k0s)**: se intentó inicialmente el despliegue mediante la distribución `k0s`, pero presentó problemas críticos de inicialización del *runtime*. Se identificó que el parámetro `cgroup_disable=memory` persistía en los parámetros del kernel incluso tras su eliminación manual en el fichero de configuración. Este comportamiento se debe a que dicho parámetro viene configurado directamente en el *Device Tree Blob* (DTB) del *firmware* de la Raspberry Pi 5, inyectándose en los parámetros del kernel y anteponiéndose a las directivas de usuario, lo que invalida la gestión de memoria por parte de ciertos orquestadores.

- **HashiCorp Nomad**: el despliegue se realizó mediante el repositorio oficial ([56]). Nomad destaca por su ligereza y su modelo de planificación desacoplado, lo que lo hace ideal para orquestar ráfagas de tareas en el Edge. A diferencia de Kubernetes, Nomad utiliza *drivers* específicos (Docker, Exec [57], Java [58]) que permiten ejecutar cargas de trabajo sin la sobrecarga de una red de pods completa si no es necesario.

  El proceso incluyó la importación de la llave GPG y la configuración de un nodo híbrido (Servidor y Cliente en el mismo dispositivo):

```
# Configuracion de APT y llave GPG oficial
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo gpg --
    dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.
    gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main"
    | sudo tee /etc/apt/sources.list.d/hashicorp.list
sudo apt update && sudo apt install nomad
```

CÓDIGO 4.4. Instalación manual de HashiCorp Nomad en Debian

La configuración principal (Código 4.5) fue crítica para permitir que Nomad mapeara volúmenes del *host* (`host_volume`) hacia los contenedores Docker, garantizando que los datos capturados por los sensores en la Raspberry Pi estuvieran disponibles para las tareas del *workflow* [7].

```
datacenter = "dc1"
data_dir   = "/opt/nomad/data"

client {
  enabled = true
  options {
    "driver.raw_exec.enable" = "1"
  }
  # Volumen para intercambio de datos con DAGonStar
  host_volume "home" {
    path      = "/home"
    read_only = false
  }
}

plugin "docker" {
  config {
    volumes { enabled = true }
    allow_privileged = false
  }
}

# Habilitacion de telemetria para Prometheus
telemetry {
  publish_allocation_metrics = true
  publish_node_metrics       = true
  prometheus_metrics         = true
}
```

CÓDIGO 4.5. Fichero de configuración /etc/nomad.d/nomad.hcl para el nodo Edge

Es fundamental distinguir entre la instalación base en el nodo Edge (que requiere una puesta a punto manual y detallada debido a las restricciones de hardware y arquitectura ARM64) y la automatización del despliegue en los nodos Fog/PC. Mientras que en la Raspberry Pi se realizó una instalación nativa para optimizar el rendimiento térmico y de almacenamiento, para los nodos de clúster operados desde el PC se utilizaron scripts de Bash reducidos. Estos scripts (Códigos 4.6 y 4.7) están diseñados específicamente para su uso en PC durante el despliegue de los workflows DAGonStar, permitiendo que el entorno esté listo para recibir tareas de cómputo pesado sin la latencia de una instalación manual

compleja.

```
#!/usr/bin/env bash
set -e
echo "[k3s] Installing..."
curl -sfL https://get.k3s.io | sh -
echo "[k3s] Configuring kubeconfig in ~/.kube/config..."
mkdir -p ~/.kube
sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/config
sudo chown $(id -u):$(id -g) ~/.kube/config
export KUBECONFIG=$HOME/.kube/config
echo "export KUBECONFIG=$HOME/.kube/config" >> ~/.bashrc
echo "[k3s] Installation completed"
```

CÓDIGO 4.6. Script de instalación de k3s

```
#!/usr/bin/env bash
set -e
echo "[Apptainer] Installing..."
if command -v apptainer &> /dev/null || command -v singularity &> /dev/
    null; then
     echo "[Apptainer] Already installed"
elif [ -f /etc/debian_version ]; then
     sudo apt update && sudo apt install -y software-properties-common
     sudo add-apt-repository -y ppa:apptainer/ppa && sudo apt update
     sudo apt install -y apptainer
elif [ -f /etc/redhat-release ]; then
     sudo dnf install -y epel-release apptainer 2>/dev/null || sudo yum
    install -y epel-release singularity
else
     curl -s https://get.apptainer.org | bash
fi
# Config environment
cat >> ~/.bashrc << EOF
export APPTAINER_CACHEDIR=\$HOME/.apptainer/cache
export APPTAINER_TMPDIR=\$HOME/.apptainer/tmp
EOF
pip install --quiet spython psutil filelock tqdm
echo "[Apptainer] Installation completed"
```

CÓDIGO 4.7. Script de instalación de Apptainer

La instalación de estas librerías de utilidad (psutil [59], filelock [60] y tqdm [61]) asegura que el entorno disponga de las herramientas necesarias para la gestión de procesos, bloqueos de archivos y visualización de progreso durante la ejecución de los experimen-

tos.

4.1.4. Métricas y herramientas

Para evaluar la solución se han definido las siguientes métricas. Los resultados completos de estas mediciones para el *workflow* IoT se encuentran disponibles para su visualización en la carpeta `/outputs` del repositorio del proyecto, consolidándose en informes estructurados:

- **Campos de Identificación y Trazabilidad (Raíz)**:
  - `_id`: identificador interno del documento en la base de datos (MongoDB).
  - `execution_id`: cadena única que identifica la ejecución del *workflow*.
  - `workflow`: nombre descriptivo del flujo de trabajo ejecutado.
  - `export_timestamp`: estampa de tiempo de la exportación de los resultados.
  - `raspberry_pi`: dirección IP del nodo Edge involucrado.
  - `raw_data_count`: contador bruto de muestras recibidas por el sistema.
  - `source_file`: ruta absoluta del fichero de datos preprocesado generado en el nodo.
  - `all_tasks_successful`: validación lógica del éxito de todas las tareas.
  - `energy_metrics_added_at`: marca temporal de la integración de métricas energéticas externas.
- **Estadísticas Ambientales (`summary`)**: resumen de la carga útil IoT.
  - `count`: número de registros válidos procesados tras el filtrado.
  - `mean_humidity` / `mean_temp_c`: promedios de humedad y temperatura.
  - `min_humidity` / `max_humidity`: extremos de humedad detectados.
  - `min_temp_c` / `max_temp_c`: extremos de temperatura detectados.
  - `start_time` / `end_time`: tiempos exactos de inicio y fin de la ventana de captura.
- **Rendimiento de Orquestación y Tareas (`metrics`)**:
  - `task_[ID]_task_duration`: duración efectiva de la tarea.
  - `task_[ID]_task_success`: resultado binario de ejecución.
  - `task_[ID]_task_records_total`: registros procesados por la tarea específica.
  - `task_[ID]_execution_start_timestamp`: momento exacto del lanzamiento de la tarea.

- `total_duration_seconds`: tiempo total de vida del flujo de trabajo (incluyendo planificación).

- **Métricas de Infraestructura y Consumo (`energy_monitoring`)**:
  - `enabled`, `prometheus_url`, `sampling_interval_seconds`: parámetros de configuración del monitor.
  - `collection_duration_seconds`, `samples_count`: métricas de salud del sistema de monitoreo.
  - Nodo Edge (Raspberry Pi) / PC (Fog/Cloud): desglosados en:
    - `power`: demanda eléctrica instantánea incluyendo la media (`mean`), picos (`max`), mínimos (`min`) y estabilidad mediante la desviación estándar (`std_watts`).
    - `cpu`: nivel de estrés del procesador expresado en porcentaje, identificando la carga media y ráfagas de computación intensiva.
    - `ram`: huella de memoria durante la ejecución, permitiendo distinguir entre el consumo base y picos de uso del orquestador.
    - `temperature`: salud térmica de la CPU en grados Celsius, crítico para detectar posibles degradaciones de rendimiento por limitación debida a exceso de temperatura.
    - `energy`: consumo acumulado durante la ráfaga, expresado en unidades de energía (Julios y Vatios-hora) para cálculos de eficiencia y coste.

La recolección se centraliza en un servidor **Prometheus** [11] que sondea los exportadores cada 2 segundos. Para la automatización del proceso, se ha desarrollado el script `energy_metrics_collector.py`, el cual interactúa con la API de Prometheus para segmentar el consumo por ráfagas de ejecución y generar informes integrados en formato JSON [36] y PDF [62]. La visualización de las métricas en tiempo real se realiza mediante cuadros de mando en **Grafana** [12].

Estructura y funcionamiento del recolector de métricas

Para automatizar la captura de consumo energético sin penalizar el rendimiento del *workflow*, se ha desarrollado la clase `PrometheusMetricsCollector`. Este módulo actúa como un agente de monitorización ligero que se ejecuta localmente en el nodo Fog, interactuando con la API de Prometheus para obtener series temporales de potencia (W), uso de CPU y temperatura.

Como se detalla en el Código 4.8, el recolector utiliza un hilo independiente (*threading*) para realizar el muestreo periódico, garantizando que la latencia de red de las peticiones HTTP no afecte a la planificación de las tareas de DAGonStar. Una de las funciones críticas es el método de integración trapezoidal, que permite estimar la energía consumida

en vatios-hora (Wh) a partir de las muestras de potencia instantánea recogidas durante el experimento.

```python
class PrometheusMetricsCollector:
    def __init__(self, prometheus_url, sampling_interval=2):
        self.url = f"{prometheus_url}/api/v1/query"
        self.interval = sampling_interval
        self.metrics_data = []

    def query_prometheus(self, query):
        """Consulta la API de Prometheus para obtener un valor escalar"""
        response = requests.get(self.url, params={'query': query})
        return float(response.json()['data']['result'][0]['value'][1])

    def _collection_loop(self):
        """Bucle de recoleccion en segundo plano"""
        while self.is_collecting:
            sample = {
                'ts': time.time(),
                'rpi_power': self.query_prometheus('rpi_pmic_power_watts'),
                'pc_power': self.query_prometheus('scaph_host_power_microwatts / 1000000')
            }
            self.metrics_data.append(sample)
            time.sleep(self.interval)

    def calculate_energy(self, power_key):
        """Calcula Joules/Wh mediante integracion trapezoidal"""
        energy = 0.0
        for i in range(len(self.metrics_data) - 1):
            dt = self.metrics_data[i+1]['ts'] - self.metrics_data[i]['ts']
            avg_p = (self.metrics_data[i][power_key] + self.metrics_data[i+1][power_key]) / 2
            energy += avg_p * dt
        return energy / 3600 # Valor final en Wh
```

CÓDIGO 4.8. Resumen de la implementación del recolector de métricas energéticas

4.1.5. Tipología de experimentos

Para cubrir los diferentes aspectos de la solución, se han diseñado tres tipos de flujos de trabajo que aumentan progresivamente en complejidad e infraestructura involucrada:

1. **Benchmark de Nodo Único (Edge)**: centrado en la eficiencia de ejecución local. El *workflow* genera ráfagas de ficheros binarios de distintos tamaños (1 KB, 1 MB, 10 MB) en la Raspberry Pi y procede a su compresión utilizando contenedores en el mismo nodo. Este escenario elimina variables de red para centrarse puramente en el *overhead* del motor de tareas y del *runtime*.

2. **Benchmark Distribuido (Edge–Fog)**: introduce la coordinación entre niveles. La generación de datos se mantiene en el Edge, pero el procesamiento (compresión) se delega al nodo Fog. Esto permite validar el mecanismo de *staging* mediante SCP y la capacidad de DAGonStar para gestionar dependencias de datos entre nodos con diferentes arquitecturas.

3. **Workflow de Telemetría IoT Completo (Edge–Fog–Cloud)**: representa un caso de uso real de monitorización ambiental. Implementa la captura de datos de un sensor DHT11 físico (Tarea A), el análisis estadístico mediante la librería Pandas [63] (Tarea B) y la persistencia de resultados en un clúster de MongoDB Atlas [16] en el Cloud (Tarea C). Este flujo se complementa con una monitorización energética exhaustiva, correlacionando el consumo de la Raspberry Pi con las fases de ejecución del contenedor.

Cabe destacar que, con el objetivo de garantizar la repetibilidad de las pruebas y centrar el análisis exclusivamente en el *overhead* de orquestación y tiempos de cómputo, todas las imágenes de contenedor requeridas para las evaluaciones han sido **precargadas** en el almacenamiento local de los nodos (Edge y Fog).

En el caso específico de **Apptainer**, debido a que la descarga de dependencias externas en el entorno de red de las pruebas resultaba insostenible por su excesiva duración (llegando a comprometer la viabilidad del experimento), se optó por utilizar una imagen personalizada que ya incluía todas las librerías necesarias para las tareas de telemetría (*pyserial [64], pandas [63], pymongo [65]*, etc.). Esta decisión metodológica explica por qué Apptainer presenta tiempos de ejecución de tarea promedio de ~20 segundos, reflejando un escenario optimizado. Por contraste, en el entorno de **Kubernetes**, la descarga incluso de imágenes base resultó ser un cuello de botella aún más crítico que en Apptainer, haciendo inviable la ejecución de ráfagas comparables bajo las mismas condiciones de red sin una preparación previa masiva, lo que supuso una solución a medias en dicho entorno.

Para los benchmarks de rendimiento introducidos en la sección anterior, se han definido dos tareas base (Tarea A y Tarea B) que ejecutan la lógica de generación y compresión

de datos mediante scripts de Bash [66]. Estas tareas se encapsulan según la tecnología de contenedores utilizada, pero mantienen la misma lógica interna para asegurar la comparabilidad de los resultados.

Tarea A: Generación de Datos (I/O-Bound)

La Tarea A tiene como objetivo primordial la creación de una carga de trabajo sintética que permita evaluar el rendimiento del subsistema de almacenamiento y la latencia de orquestación. El script, implementado en Bash [66], genera de forma serializada 10 ficheros binarios distribuyendo tres tamaños predefinidos (1 KB, 1 MB y 10 MB) para simular distintos perfiles de tráfico IoT (desde simples lecturas de sensores hasta ráfagas de telemetría más pesadas).

Como se observa en el Código 4.9, el script utiliza el comando `dd` con la fuente `/dev/urandom`, lo que garantiza que los datos generados no sean trivialmente compresibles por el sistema de archivos, forzando así un uso real de los recursos. Además, implementa un sistema de gestión de logs y captura de métricas en formato JSON [36], lo que permite al motor de *workflow* y a las herramientas de análisis posterior conocer la duración exacta de la fase de escritura independientemente del *overhead* del contenedor.

```
TASK="A"
WORKFLOW="Edge-Only-File-Processing"
EXECUTION_ID="${EXECUTION_ID}"

NUM_FILES=10
OUTPUT_DIR="/home/raspi/edge_only_files_${EXECUTION_ID}"
SIZES=(1024 1048576 10485760)
SIZE_NAMES=("1KB" "1MB" "10MB")

START_TS=$(date +%s.%N)
SUCCESS=1
FILES_GENERATED=0

LOG_FILE="/home/raspi/edge_only_taskA_${EXECUTION_ID}.log"
exec > >(tee -a "$LOG_FILE") 2>&1

echo "========================================="
echo "Starting Task A: File Generation (EDGE)"
echo "Execution ID: ${EXECUTION_ID}"
echo "Node: Raspberry Pi (Edge)"
echo "========================================="

# Creacion del directorio de salida
mkdir -p "${OUTPUT_DIR}"
```

```
25
26  # Generacion de ficheros con dd
27  for i in $(seq 1 $NUM_FILES); do
28      size_idx=$(((i - 1) % 3))
29      size=${SIZES[$size_idx]}
30      size_name=${SIZE_NAMES[$size_idx]}
31      filename="${OUTPUT_DIR}/file_${i}_${size_name}.dat"
32
33      echo "Generating file $i of $NUM_FILES (size: ${size_name})..."
34
35      if dd if=/dev/urandom of="$filename" bs="$size" count=1 status=none
        2>/dev/null; then
36          echo "  Created: $filename"
37          FILES_GENERATED=$((FILES_GENERATED + 1))
38      else
39          echo "  ERROR creating: $filename"
40          SUCCESS=0
41      fi
42  done
43
44  END_TS=$(date +%s.%N)
45  DURATION=$(awk "BEGIN {print ${END_TS}-${START_TS}}")
46
47  # Guardado de metricas locales
48  METRICS_FILE="/home/raspi/edge_only_taskA_metrics_${EXECUTION_ID}.json"
49  cat > "${METRICS_FILE}" << EOF
50  {
51    "task": "A",
52    "execution_id": "${EXECUTION_ID}",
53    "duration_seconds": ${DURATION},
54    "success": ${SUCCESS},
55    "files_generated": ${FILES_GENERATED}
56  }
57  EOF
```

CÓDIGO 4.9. Código resumido del script de la Tarea A (Generación)

Tarea B: Procesamiento y Compresión (CPU-Bound)

La Tarea B representa la fase de computación intensiva del benchmark. Su función es leer los ficheros de datos en bruto producidos por la tarea anterior y aplicarles una compresión mediante la herramienta `gzip` [67]. Este proceso es especialmente relevante en el Edge Computing, donde la compresión previa al envío es una estrategia común para ahorrar ancho de banda, pero a costa de un mayor uso de CPU y generación de calor en

el dispositivo.

El script (Código 4.10) incluye mecanismos de verificación de pre-condiciones, como la existencia del directorio de entrada y la disponibilidad de las herramientas de compresión. Durante la ejecución, calcula el ratio de compresión en tiempo real para validar la integridad del proceso y, al igual que la Tarea A, genera un informe de métricas final. La medición de la duración en esta tarea es crítica para identificar la limitación de rendimiento por exceso de temperatura en la Raspberry Pi tras periodos de carga sostenida.

```
TASK="B"
WORKFLOW="Edge-Only-File-Processing"
EXECUTION_ID="${EXECUTION_ID}"

INPUT_DIR="/home/raspi/edge_only_files_${EXECUTION_ID}"
OUTPUT_DIR="/home/raspi/edge_only_compressed_${EXECUTION_ID}"
SIZE_NAMES=("1KB" "1MB" "10MB")

START_TS=$(date +%s.%N)
SUCCESS=1
FILES_COMPRESSED=0

LOG_FILE="/home/raspi/edge_only_taskB_${EXECUTION_ID}.log"
exec > >(tee -a "$LOG_FILE") 2>&1

echo "========================================="
echo "Starting Task B: File Compression (EDGE)"
echo "Execution ID: ${EXECUTION_ID}"
echo "========================================="

# Verificacion de directorio de entrada
if [ ! -d "${INPUT_DIR}" ]; then
    echo "ERROR: Input directory not found: ${INPUT_DIR}"
    exit 1
fi
mkdir -p "${OUTPUT_DIR}"

# Procesado y compresion
for i in $(seq 1 10); do
    size_idx=$(((i - 1) % 3))
    size_name=${SIZE_NAMES[$size_idx]}
    input_file="${INPUT_DIR}/file_${i}_${size_name}.dat"
    output_file="${OUTPUT_DIR}/file_${i}_${size_name}.gz"

    echo "Compressing file $i of 10..."
    if gzip -c "$input_file" > "$output_file" 2>&1; then
```

```
        original_size=$(stat -c%s "$input_file")
        compressed_size=$(stat -c%s "$output_file")
        compression_ratio=$(awk "BEGIN {printf \"%.2f\", (
    $original_size - $compressed_size) * 100 / $original_size}")
        echo "  Compressed (${compression_ratio}% reduction)"
        FILES_COMPRESSED=$((FILES_COMPRESSED + 1))
    else
        echo "  ERROR in gzip"
        SUCCESS=0
    fi
done

END_TS=$(date +%s.%N)
DURATION=$(awk "BEGIN {print ${END_TS}-${START_TS}}")

# Guardado de metricas
METRICS_FILE="/home/raspi/edge_only_taskB_metrics_${EXECUTION_ID}.json"
cat > "${METRICS_FILE}" << EOF
{
  "task": "B",
  "execution_id": "${EXECUTION_ID}",
  "duration_seconds": ${DURATION},
  "success": ${SUCCESS},
  "files_compressed": ${FILES_COMPRESSED}
}
EOF
```

CÓDIGO 4.10. Código resumido del script de la Tarea B (Compresión)

4.1.6. Implementación de Benchmarks con DAGonStar

La integración de estos scripts en el motor de *workflow* se realiza mediante la instanciación de clases específicas en Python [18] según el orquestador o *runtime*. A continuación se presentan los ejemplos de configuración para cada tecnología evaluada, asumiendo en todos los casos que las variables `taskA_command` y `taskB_command` contienen la lógica Bash detallada anteriormente para asegurar la comparabilidad de los resultados.

Ejemplos de Configuración por Tecnología

1. **Docker**: Se especifica el directorio de trabajo dentro del contenedor y el montaje de volúmenes para asegurar la persistencia de los ficheros generados.

```
import configparser, time
```

```
from dagon import Workflow
from dagon.task import DagonTask, TaskType

config = configparser.ConfigParser()
config.read('dagon.ini')
RASPI_IP = config.get('ssh', 'raspi_ip')
USER = config.get('ssh', 'raspi_user')

workflow = Workflow("Edge-Only-Docker")

taskA = DagonTask(TaskType.DOCKER, "A", taskA_command,
                  image="ubuntu:22.04", ip=RASPI_IP,
                  ssh_username=USER,
                  working_dir=f"/home/{USER}/scratch/taskA",
                  volume=f"/home/{USER}:/home/{USER}")

taskB = DagonTask(TaskType.DOCKER, "B", taskB_command,
                  image="ubuntu:22.04", ip=RASPI_IP,
                  ssh_username=USER,
                  working_dir=f"/home/{USER}/scratch/taskB",
                  volume=f"/home/{USER}:/home/{USER}")

taskB.add_dependency_to(taskA)
workflow.add_tasks([taskA, taskB])
workflow.run()
```

CÓDIGO 4.11. Workflow resumido con Docker

2. **Apptainer**: Se añaden los `extra_args` necesarios para habilitar la escritura en directorios temporales y el mapeo de rutas del host.

```
import configparser, time
from dagon import Workflow
from dagon.task import DagonTask, TaskType

config = configparser.ConfigParser()
config.read('dagon.ini')
RASPI_IP = config.get('ssh', 'raspi_ip')
USER = config.get('ssh', 'raspi_user')

workflow = Workflow("Edge-Only-Apptainer")

taskA = DagonTask(TaskType.APPTAINER, "A", taskA_command,
                  image="docker://ubuntu:22.04", ip=RASPI_IP,
                  ssh_username=USER)
```

```
taskA.extra_args = ["--bind", f"/home/{USER}:/home/{USER}",
                    "--writable-tmpfs"]

taskB = DagonTask(TaskType.APPTAINER, "B", taskB_command,
                  image="docker://ubuntu:22.04", ip=RASPI_IP,
                  ssh_username=USER)
taskB.extra_args = ["--bind", f"/home/{USER}:/home/{USER}",
                    "--writable-tmpfs"]

taskB.add_dependency_to(taskA)
workflow.add_tasks([taskA, taskB])
workflow.run()
```

CÓDIGO 4.12. Workflow resumido con Apptainer

3. **Kubernetes**: Se define la ejecución sobre el cluster k3s subyacente, delegando la gestión de pods al orquestador.

```
import configparser, time
from dagon import Workflow
from dagon.task import DagonTask, TaskType

config = configparser.ConfigParser()
config.read('dagon.ini')
RASPI_IP = config.get('ssh', 'raspi_ip')

workflow = Workflow("Edge-Only-Kubernetes")

taskA = DagonTask(TaskType.KUBERNETES, "A", taskA_command,
                  image="alpine:latest", ip=RASPI_IP,
                  working_dir="/scratch",
                  volume="/mnt/data:/scratch")

taskB = DagonTask(TaskType.KUBERNETES, "B", taskB_command,
                  image="alpine:latest", ip=RASPI_IP,
                  working_dir="/scratch",
                  volume="/mnt/data:/scratch")

taskB.add_dependency_to(taskA)
workflow.add_tasks([taskA, taskB])
workflow.run()
```

CÓDIGO 4.13. Workflow resumido con Kubernetes

4. **HashiCorp Nomad**: Ejemplo de control granular, limitando la carga de CPU y

memoria para evitar inestabilidad en el dispositivo Edge.

```
import configparser, time
from dagon import Workflow
from dagon.task import DagonTask, TaskType

config = configparser.ConfigParser()
config.read('dagon.ini')
RASPI_IP = config.get('ssh', 'raspi_ip')
USER = config.get('ssh', 'raspi_user')
NOMAD_ADDR = config.get('nomad', 'address')

workflow = Workflow("Edge-Only-Nomad")

taskA = DagonTask(TaskType.NOMAD, "A", taskA_command,
                  image="ubuntu:22.04", nomad_address=NOMAD_ADDR,
                  ip=RASPI_IP, ssh_username=USER,
                  working_dir=f"/home/{USER}/nomad_a",
                  cpu=200, memory=512)

taskB = DagonTask(TaskType.NOMAD, "B", taskB_command,
                  image="ubuntu:22.04", nomad_address=NOMAD_ADDR,
                  ip=RASPI_IP, ssh_username=USER,
                  working_dir=f"/home/{USER}/nomad_b",
                  cpu=200, memory=512)

taskB.add_dependency_to(taskA)
workflow.add_tasks([taskA, taskB])
workflow.run()
```

CÓDIGO 4.14. Workflow resumido con Nomad

Estos ejemplos corresponden al escenario *Edge-Only*, donde la orquestación y el procesamiento ocurren íntegramente en el nodo Raspberry Pi. No obstante, en el escenario híbrido **Edge-Fog**, se introduce una fase crítica de movimiento de datos (staging) entre la finalización de la Tarea A (Edge) y el inicio de la Tarea B (Fog).

En esta arquitectura distribuida, la Tarea B se configura para ejecutarse en el nodo local (la estación base o Fog node), eliminando los parámetros `ip` y `ssh_username` de su definición. Para garantizar que los datos generados en el dispositivo de borde estén disponibles en el nodo de procesamiento superior, el script de orquestación implementa una transferencia mediante el protocolo SCP (*Secure Copy*), tal como se detalla en el Código 4.15.

```
import subprocess, os
```

```
2
3   # Configuracion de rutas
4   fog_input_dir = f"/tmp/fog_input_{EXECUTION_ID}"
5   edge_files_dir = f"/home/{USER}/edge_files_{EXECUTION_ID}"
6   os.makedirs(fog_input_dir, exist_ok=True)
7
8   # Comando SCP para la descarga recursiva
9   scp_cmd = [
10      'scp', '-r', '-P', str(RASPI_PORT),
11      f'{USER}@{RASPI_IP}:{edge_files_dir}/*',
12      f'{fog_input_dir}/'
13  ]
14
15  # Ejecucion sincronica de la transferencia
16  try:
17      result = subprocess.run(scp_cmd, capture_output=True, check=True)
18      print(f"Transferencia completada: {len(os.listdir(fog_input_dir))}
        ficheros")
19  except subprocess.CalledProcessError as e:
20      print(f"Error en el staging: {e.stderr}")
```

CÓDIGO 4.15. Transferencia de archivos Edge-to-Fog mediante SCP

Este mecanismo de *staging* asegura que la Tarea B, al ejecutarse en un entorno con mayores recursos de CPU, pueda acceder a la carga de trabajo de forma local, minimizando la latencia de acceso a disco durante la fase de compresión comparado con un acceso remoto a través de la red.

4.1.7. Implementación y Configuración del Workflow IoT

El despliegue del flujo de telemetría se apoya en una configuración centralizada en el fichero `dagon.ini`, donde se definen los parámetros de conexión para los tres niveles de la arquitectura *edge-to-cloud*:

- **Nivel Edge**: IP y credenciales SSH de la Raspberry Pi, junto con los *endpoints* de los orquestadores ligeros (Nomad/k3s).
- **Nivel Fog**: configuración del servidor local de Prometheus y el *token* de la API de Grafana para la exportación de resultados.
- **Nivel Cloud**: URI de conexión a la instancia de **MongoDB Atlas** [16], incluyendo la base de datos (`IoT_Metrics`) y la colección de destino.

La lógica de control se implementa mediante scripts de Python que utilizan la librería DAGon. Es importante destacar que una de las ventajas fundamentales de la solución

propuesta es que el código de las tareas es **agnóstico a la tecnología de contenedores**. Gracias a la abstracción de DAGonStar [1], [2], el mismo script de Bash o Python se ejecuta de forma idéntica ya sea sobre Docker, Apptainer o Nomad, permitiendo una portabilidad total del flujo de trabajo sin necesidad de modificar la lógica de negocio.

El flujo de ejecución sigue el patrón de tres tareas interconectadas: la **Tarea A** ejecuta un loop de lectura de 300 segundos sobre el sensor serie; la **Tarea B** utiliza la librería *Pandas* [63] para calcular valores medios, máximos y mínimos de temperatura y humedad; y finalmente la **Tarea C** unifica estas métricas funcionales con los datos de consumo energético obtenidos de Prometheus, insertando un documento único en el Cloud que permite la trazabilidad completa del experimento. En la Figura 4.4 se detalla la secuencia de operaciones y el flujo de datos entre los distintos componentes del motor durante la ejecución de este workflow.

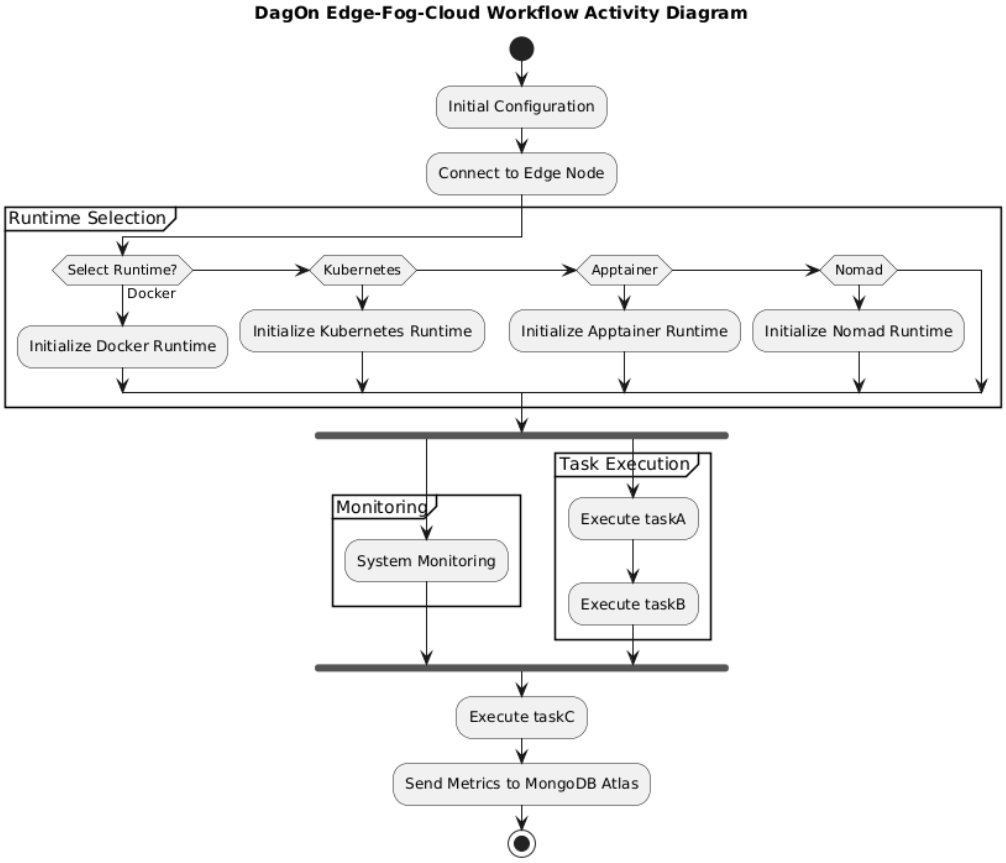


Figura 4.4
Diagrama de actividad del workflow IoT con DAGonStar.

Tarea A: Captura de Datos del Sensor (Edge)

La Tarea A se sitúa en el extremo más cercano al origen del dato (Edge). Su función es orquestar la comunicación con el sensor físico DHT11 y asegurar que las lecturas se capturen de forma ininterrumpida durante la ventana temporal del experimento. Como se

detalla en el Código 4.16, la tarea encapsula un pequeño programa Python dentro de una estructura de control Bash que gestiona la instalación dinámica de dependencias (como `pyserial` [64]) y la generación de métricas en formato Prometheus.

```
TASK="A"
WORKFLOW="DHT-Sensor-Capture-And-Preprocess"
EXECUTION_ID="${EXECUTION_ID}"

METRICS_FILE="/home/${RASPI_USER}/.dagon_metrics/history/dagon_dht_A_${EXECUTION_ID}.prom"

START_TS=$(date +%s.%N)
SUCCESS=1
RECORDS=0

# Logica de instalacion de dependencias y toma de datos
pip3 install --quiet --no-cache-dir pyserial || true

python3 << 'EOF'
import serial, re, json, time, os
PORT = '/dev/ttyACM0'
BAUDRATE = 9600
DURATION = 300
OUTPUT_FILE = 'output.json'

try:
    ser = serial.Serial(PORT, BAUDRATE, timeout=2)
    data, start = [], time.time()
    while time.time() - start < DURATION:
        line = ser.readline().decode('utf-8', errors='ignore').strip()
        m = re.search(r'Humidity:\s*([\d.]+).*Temperature:\s*([\d.]+)', line)
        if m:
            data.append({'timestamp': time.time(), 'humidity': float(m[1]), 'temp_c': float(m[2])})
    ser.close()
    print(f"RECORDS_CAPTURED={len(data)}")
    json.dump(data, open(OUTPUT_FILE, 'w'), indent=2)
except Exception as e:
    print(f"ERROR: {e}")
    exit(1)
EOF

# Reporte de métricas unificadas (formato Prometheus)
cat > ${METRICS_FILE} << EOF_METRICS
```

```
dagon_task_duration_seconds{workflow="IoT",task="A",execution_id="${
    EXECUTION_ID}"} ${DURATION}
dagon_task_success{workflow="IoT",task="A",execution_id="${EXECUTION_ID
    }"} ${SUCCESS}
dagon_task_records_total{workflow="IoT",task="A",execution_id="${
    EXECUTION_ID}"} ${RECORDS}
EOF_METRICS
```

CÓDIGO 4.16. Script de ejecución de la Tarea A (Captura de datos)

La particularidad de esta tarea reside en su capacidad para operar sobre hardware real utilizando contenedores. El motor de *workflow* debe garantizar que el dispositivo serie sea accesible desde el entorno aislado, lo que se resuelve mediante el mapeo de volúmenes o privilegios específicos en el orquestador correspondiente. Al finalizar la captura, la tarea genera un fichero JSON [36] con la serie temporal bruta, que será la entrada de la siguiente fase.

Tarea B: Preprocesamiento Estadístico (Fog)

La Tarea B asume el rol de procesamiento intermedio, típicamente ubicado en el nivel Fog para liberar de carga computacional al dispositivo Edge. Su objetivo es transformar la serie temporal de lecturas en un resumen estadístico significativo. Como se observa en el Código 4.17, esta tarea hace un uso intensivo de la librería *Pandas* [63] para calcular agregados de temperatura y humedad.

```
TASK="B"
EXECUTION_ID="${EXECUTION_ID}"
DATA_DIR="/tmp/fog_data_${EXECUTION_ID}"
INPUT_FILE="${DATA_DIR}/sensor_output_${EXECUTION_ID}.json"
OUTPUT_FILE="${DATA_DIR}/sensor_output_${EXECUTION_ID}_preprocessed.
    json"

# Instalacion de dependencias de analisis
pip3 install --quiet --no-cache-dir pandas || true

python3 << 'PYEOF'
import json, os
import pandas as pd

with open(INPUT_FILE) as f:
    data = json.load(f)

df = pd.DataFrame(data)
summary = {
```

```
    "count": int(len(df)),
    "mean_humidity": float(df["humidity"].mean()),
    "mean_temp_c": float(df["temp_c"].mean()),
    "min_humidity": float(df["humidity"].min()),
    "max_humidity": float(df["humidity"].max()),
    "min_temp_c": float(df["temp_c"].min()),
    "max_temp_c": float(df["temp_c"].max())
}

with open(OUTPUT_FILE, 'w') as out:
    json.dump({"execution_id": "${EXECUTION_ID}", "summary": summary, "
    raw": data}, out, indent=2)
PYEOF
```

CÓDIGO 4.17. Script de ejecución de la Tarea B (Preprocesamiento estadístico)

Esta fase es crítica para la eficiencia del *workflow* completo, ya que reduce el volumen de información que debe enviarse al nivel Cloud. En lugar de persistir miles de lecturas individuales, el sistema genera un documento estructurado que contiene tanto los datos crudos como el análisis derivado, facilitando la visualización posterior en Grafana. La ejecución en el Fog permite aprovechar una mayor capacidad de cómputo y memoria que la disponible en la Raspberry Pi.

Tarea C: Persistencia en la Nube (Cloud)

La Tarea C constituye el punto final del flujo de datos, encargándose de la persistencia en un clúster gestionado de MongoDB Atlas [16]. Su función principal es unificar el conocimiento generado en las etapas anteriores: los datos estadísticos de la Tarea B y las métricas de rendimiento (duración, éxito, registros) de ambas tareas (A y B). Como se detalla en el Código 4.18, la tarea utiliza la librería `pymongo` [65] y realiza peticiones remotas para consolidar la información.

```
TASK="C"
EXECUTION_ID="${EXECUTION_ID}"
DATA_DIR="/tmp/fog_data_${EXECUTION_ID}"
INPUT_FILE="${DATA_DIR}/sensor_output_${EXECUTION_ID}_preprocessed.json
    "

# Instalacion de dependencias de persistencia
pip3 install --quiet --no-cache-dir pymongo || true

python3 << 'PYEOF'
import json, re, os, subprocess
from pymongo import MongoClient
```

```
from datetime import datetime, timezone

# Conexion con MongoDB Atlas y preparacion del documento
client = MongoClient("${MONGO_URI}")
db = client["${MONGO_DB}"]
collection = db["${MONGO_COLLECTION}"]

# Recoleccion de metricas de la Tarea A via SCP desde el Edge
edge_metrics = "/home/${RASPI_USER}/.dagon_metrics/history/dht_A_${
    EXECUTION_ID}.prom"
local_metrics = f"{DATA_DIR}/taskA_metrics.prom"
subprocess.run(['scp', '-P', '${RASPI_PORT}', f'${RASPI_USER}@${
    RASPI_IP}:{edge_metrics}', local_metrics])

# Construccion del documento final consolidado
document = {
    "execution_id": "${EXECUTION_ID}",
    "workflow": "DHT-Sensor-Capture-And-Preprocess",
    "export_timestamp": datetime.now(timezone.utc),
    "summary": data.get("summary", {}),
    "metrics": collect_all_metrics(),
    "all_tasks_successful": check_success()
}
collection.insert_one(document)
PYEOF
```

CÓDIGO 4.18. Script de ejecución de la Tarea C (Exportación a MongoDB Atlas)

La implementación de esta tarea destaca por su enfoque integrador. Al realizar un `scp` para traer las métricas de la Tarea A desde el Edge al Fog (donde reside la Tarea C), se logra una trazabilidad completa de la ejecución sin necesidad de que el nodo Edge tenga acceso directo a la base de datos Cloud, mejorando así la seguridad y segmentación de la red. El documento resultante permite realizar consultas complejas sobre la eficiencia energética y el rendimiento funcional del sistema en una única plataforma de observabilidad.

Implementación Completa de los Workflows de Telemetría

Una vez definida la lógica interna de las tareas, la orquestación recae en scripts de Python que gestionan no solo la ejecución paralela, sino también la captura de métricas energéticas y la sincronización entre niveles. A continuación se presentan los scripts completos para cada tecnología, omitiendo las variables de comando detalladas en las secciones anteriores para centrarse en el flujo de control.

Caso 1: Orquestación con Docker: El despliegue con Docker utiliza una estrategia de tres fases: ejecución en el Edge (Tarea A), transferencia de ficheros mediante SCP (Staging) y ejecución en el Fog (Tareas B y C). El Código 4.19 muestra cómo se integran las clases de DAGon con el recolector de métricas de Prometheus.

```
import json, configparser, logging, time
from dagon import Workflow
from dagon.task import DagonTask, TaskType
from datetime import datetime, timezone
from pymongo import MongoClient
import sys, os, subprocess

# Recolector de metricas de energia (clase custom descrita en seccion 4.1)
from energy_metrics_collector import PrometheusMetricsCollector

# Lectura de configuracion centralizada
config = configparser.ConfigParser()
config.read('dagon.ini')

RASPI_IP = config.get('ssh', 'raspi_ip')
RASPI_USER = config.get('ssh', 'raspi_user')
RASPI_PORT = config.getint('ssh', 'raspi_port', fallback=22)

MONGO_URI = config.get('mongodb', 'uri')
MONGO_DB = config.get('mongodb', 'database')
MONGO_COLLECTION = config.get('mongodb', 'collection')
PROMETHEUS_URL = config.get('prometheus', 'url', fallback='http://localhost:9090')
SAMPLING_INTERVAL = config.getint('prometheus', 'sampling_interval', fallback=2)

EXECUTION_ID = time.strftime("%Y%m%d_%H%M%S")

# Definicion de variables de tarea (BASH) - Ver códigos 4.1-4.3
taskA_command = "# ... (codigo detallado anteriormente) ..."
taskB_command = "# ... (codigo detallado anteriormente) ..."
taskC_command = "# ... (codigo detallado anteriormente) ..."

def execute_with_transfer():
    # FASE 1: Ejecucion en Edge (Raspberry Pi)
    taskA = DagonTask(TaskType.DOCKER, "A", taskA_command,
                      image="python:3.9-slim", ip=RASPI_IP,
                      ssh_username=RASPI_USER, ssh_port=RASPI_PORT,
                      working_dir=f"/home/{RASPI_USER}/scratch_docker/
```

```
    taskA",
                        volume=f"/home/{RASPI_USER}:/home/{RASPI_USER}",
                        devices=["/dev/ttyACM0:/dev/ttyACM0"])
    wf_a = Workflow("Task-A")
    wf_a.add_task(taskA)
    wf_a.run()

    # FASE 2: Movimiento de datos (Staging local)
    local_file = f"/tmp/fog_data_{EXECUTION_ID}/sensor_output_{
    EXECUTION_ID}.json"
    edge_file = f"/home/{RASPI_USER}/sensor_output_{EXECUTION_ID}.json"
    subprocess.run(['scp', '-P', str(RASPI_PORT),
                    f'{RASPI_USER}@{RASPI_IP}:{edge_file}', local_file
    ], check=True)

    # FASE 3: Ejecucion en Fog (PC local)
    taskB = DagonTask(TaskType.DOCKER, "B", taskB_command,
                        image="python:3.9-slim", volume="/tmp:/tmp")
    taskC = DagonTask(TaskType.DOCKER, "C", taskC_command,
                        image="python:3.9-slim", volume="/tmp:/tmp")
    taskC.add_dependency_to(taskB)
    wf_bc = Workflow("Tasks-BC")
    wf_bc.add_tasks([taskB, taskC])
    wf_bc.run()

# Bloque principal de monitorizacion
collector = PrometheusMetricsCollector(PROMETHEUS_URL,
    SAMPLING_INTERVAL)
collector.start_collection()
try:
    execute_with_transfer()
finally:
    collector.stop_collection()
    stats = collector.calculate_statistics()
    # ... Logica de actualizacion en MongoDB (identica para todos los
    casos) ...
```

CÓDIGO 4.19. Workflow resumido de telemetría IoT sobre Docker

Caso 2: Orquestación con Nomad: En el caso de Nomad, el orquestador gestiona directamente el despliegue en el nodo remoto. El script simplifica el movimiento de datos al apoyarse en la infraestructura de Nomad para el control de recursos y la visibilidad de los logs del cluster a través de una dirección IP de gestión centralizada (`nomad_address`).

```python
# ... (Imports y configuracion identica al Caso 1) ...
EDGE_NOMAD_URL = "http://192.168.1.100:4646"

def run_nomad_workflow():
    # Definicion de tareas bajo el orquestador Nomad
    taskA = DagonTask(TaskType.NOMAD, "A", taskA_command,
                      image="python:3.9-slim", nomad_address=
    EDGE_NOMAD_URL,
                      ip=RASPI_IP, ssh_username=RASPI_USER,
                      cpu=250, memory=256) # Reserva explicita de
    recursos

    taskB = DagonTask(TaskType.NOMAD, "B", taskB_command,
                      image="python:3.9-slim", nomad_address="http://
    localhost:4646",
                      volume="/tmp:/tmp", cpu=500, memory=512)

    taskC = DagonTask(TaskType.NOMAD, "C", taskC_command,
                      image="python:3.9-slim", nomad_address="http://
    localhost:4646",
                      volume="/tmp:/tmp")

    taskB.add_dependency_to(taskA)
    taskC.add_dependency_to(taskB)
    workflow = Workflow("IoT-Nomad-Workflow")
    workflow.add_tasks([taskA, taskB, taskC])
    workflow.run()

# Ejecucion con monitorizacion integrada
collector.start_collection()
run_nomad_workflow()
# ... actualizacion de resultados en la Nube ...
```

CÓDIGO 4.20. Workflow resumido de telemetría IoT sobre Nomad

Caso 3: Orquestación con Apptainer: El script para Apptainer destaca por el uso de imágenes `.sif` y una integración profunda con el sistema de archivos del host mediante parámetros de montaje (`-bind`). Este enfoque permite un rendimiento cercano al nativo pero con el aislamiento y reproducibilidad de los contenedores.

```python
# ... (Imports y configuracion identica al Caso 1) ...

workflow = Workflow("Apptainer-IoT-Workflow")
```

```
4
5  taskA = DagonTask(TaskType.APPTAINER, "A", taskA_command,
6                    image="/home/raspi/python_dht.sif", ip=RASPI_IP,
7                    ssh_username=RASPI_USER)
8  taskA.extra_args = ["--bind", "/dev/ttyACM0:/dev/ttyACM0",
9                      "--bind", "/home/raspi:/home/raspi",
10                     "--bind", "/var/lib/node_exporter:/var/lib/
       node_exporter:rw",
11                     "--writable-tmpfs"]
12
13 taskB = DagonTask(TaskType.APPTAINER, "B", taskB_command,
14                   image="/home/raspi/python_dht.sif", ip=RASPI_IP,
15                   ssh_username=RASPI_USER)
16 taskB.extra_args = ["--bind", "/home/raspi:/home/raspi", "--writable-
       tmpfs"]
17
18 # ... definicion de dependencias y ejecucion ...
19 workflow.add_tasks([taskA, taskB, taskC])
20 workflow.run()
```

CÓDIGO 4.21. Workflow resumido de telemetría IoT sobre Apptainer

4.1.8. Gestión de Datos y Monitorización en MongoDB Atlas

La persistencia del flujo de telemetría IoT se apoya en un clúster de **MongoDB Atlas** desplegado en la región de AWS [68] Paris (*eu-west-3*). La elección de una base de datos NoSQL documental facilita el almacenamiento de métricas heterogéneas provenientes de distintos orquestadores, permitiendo evolucionar el esquema de datos sin necesidad de migraciones complejas.

El motor de *workflow* gestiona la autenticación mediante el uso de secretos inyectados en las tareas, asegurando una conexión estable y cifrada con el *replica set* del clúster Cloud. La monitorización del estado del clúster se realiza a través del panel de control de Atlas, donde se supervisan indicadores clave como el número de conexiones activas, el flujo de datos entrante y saliente, y latencias de operación (Figura 4.5).

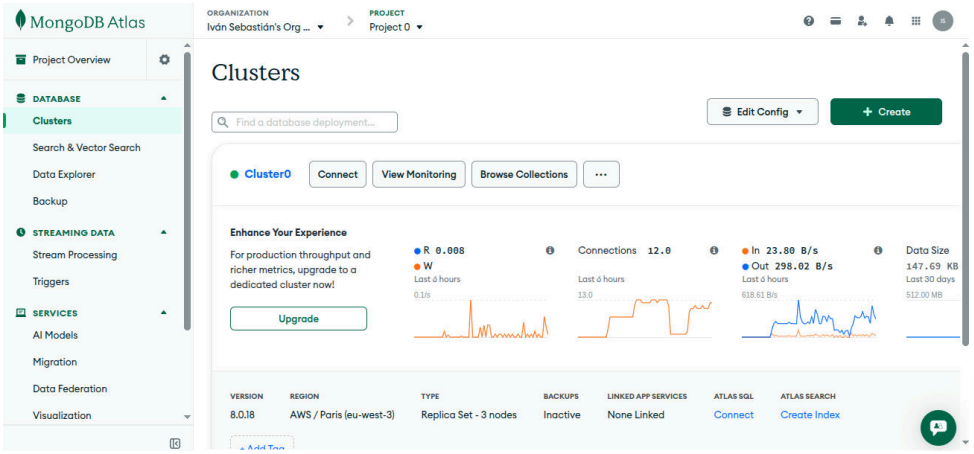


Figura 4.5

Dashboard de MongoDB Atlas mostrando la vista general del cluster (Cluster0). Se observan métricas de rendimiento en tiempo real y la configuración del Replica Set.

Fuente: elaboración propia a partir de la consola de administración de MongoDB Atlas.

Para garantizar la estabilidad del servicio durante las ráfagas de pruebas, se analizan los contadores de operaciones (*opcounters*), que permiten verificar que la carga de trabajo inducida por las tareas del *workflow* se distribuye correctamente entre los nodos del clúster (Figura 4.6).

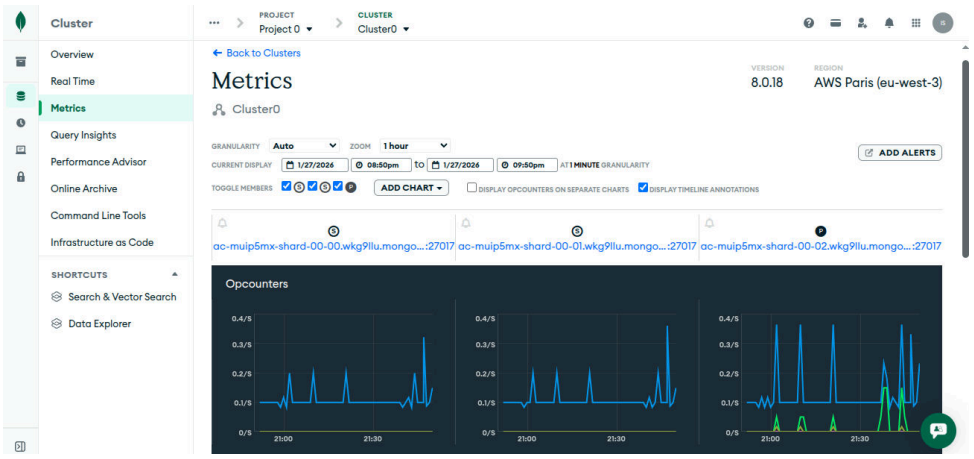


Figura 4.6

Métricas de rendimiento del cluster en MongoDB Atlas. Visualización temporal de los contadores de operaciones para los nodos del replica set.

Fuente: elaboración propia.

La organización lógica de la información se estructura en la base de datos `IoT_Metrics`, donde se han definido colecciones independientes para cada tecnología de contenedor evaluada (`Apptainer_Metrics`, `Docker_Metrics` y `Nomad_Metrics`). Esta separación facilita el análisis comparativo posterior y permite gestionar índices específicos según los volúmenes de datos generados por cada orquestador (Figura 4.7).

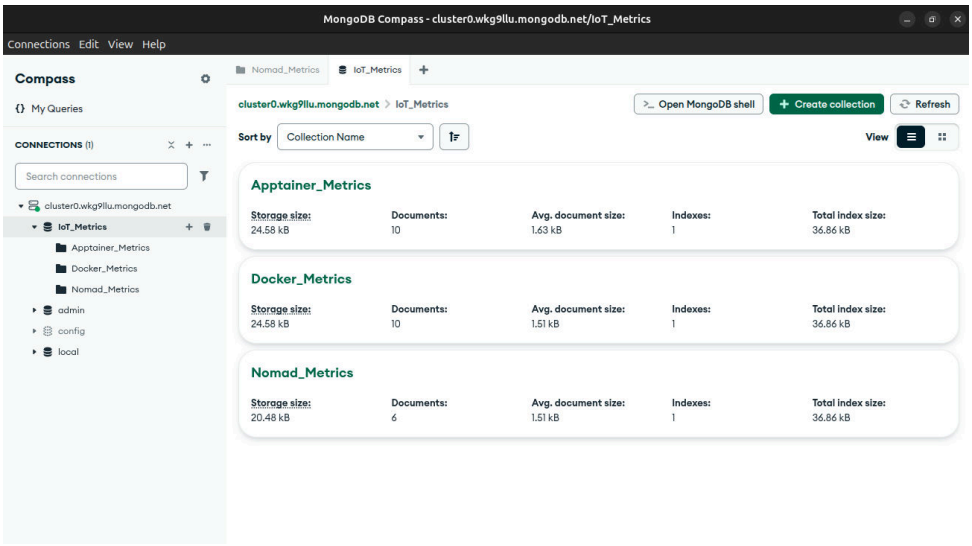


Figura 4.7
Código de colecciones de la base de datos `IoT_Metrics` *visualizadas en MongoDB Compass.*

Fuente: elaboración propia.

Cada documento insertado en las colecciones captura el contexto completo de una ejecución, incluyendo identificadores únicos, marca de tiempo, métricas del sensor y los resultados del análisis estadístico. En la Figura 4.8 se muestra la estructura JSON de uno de estos documentos generados tras una ejecución con Apptainer, destacando la jerarquía de campos que permite navegar desde los datos brutos hasta los agregados.

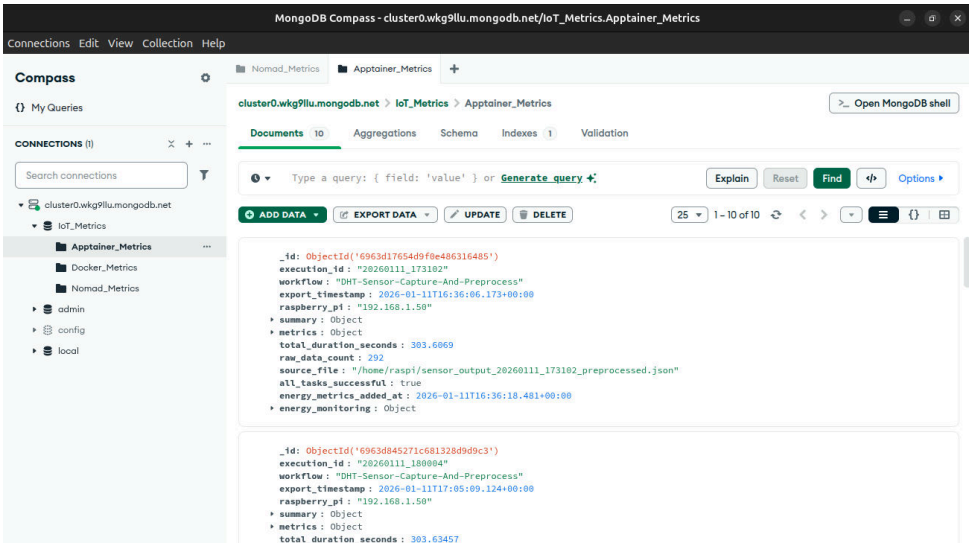


Figura 4.8
Documentos JSON dentro de la colección `Apptainer_Metrics` *mostrando el esquema de datos utilizado para la persistencia.*

Fuente: elaboración propia.

Esquema de Datos Unificado para Observabilidad

Para permitir un análisis post-ejecución profundo, se ha diseñado un esquema de datos no relacional basado en el estándar JSON [69] que consolida tres niveles de información en un único documento. Esta estructura implementa de forma exhaustiva los campos y métricas analizados detalladamente en la Sección 4.1.4. El Código 4.22 muestra un ejemplo real de esta estructura para una ejecución de telemetría IoT.

- **Bloque `summary`**: contiene el estado funcional de la aplicación. En este caso, el número de muestras procesadas y los promedios de temperatura y humedad capturados por el sensor físico.
- **Bloque `metrics`**: registra el estado del motor de *workflow*. Incluye la duración exacta de cada tarea (A y B), el éxito de la operación y las marcas de tiempo de inicio, permitiendo calcular latencias de orquestación.
- **Bloque `energy_monitoring`**: representa el estado fisiológico de la infraestructura. Integra métricas segmentadas para el nodo Edge (Raspberry Pi) y el nodo Fog (PC), correlacionando el uso de CPU y temperatura con el consumo energético en Joules y Vatios-hora (Wh).


```
[{
  "_id": {
    "$oid": "6963d17654d9f0e486316485"
  },
  "execution_id": "20260111_173102",
  "workflow": "DHT-Sensor-Capture-And-Preprocess",
  "export_timestamp": {
    "$date": "2026-01-11T16:36:06.173Z"
  },
  "raspberry_pi": "192.168.1.50",
  "summary": {
    "count": 292,
    "mean_humidity": 60,
    "mean_temp_c": 17.611643835616434,
    "min_humidity": 60,
    "max_humidity": 60,
    "min_temp_c": 17.6,
    "max_temp_c": 17.7,
    "start_time": 1768149056.7368157,
    "end_time": 1768149357.6128218
  },
  "metrics": {
    "task_A_task_duration": 301.003,
```

```
      "task_A_task_success": 1,
      "task_A_task_records_total": 292,
      "task_A_execution_start_timestamp": 1768149056.6393135,
      "task_B_task_duration": 2.6039,
      "task_B_task_success": 1,
      "task_B_task_records_total": 292,
      "task_B_execution_start_timestamp": 1768149360.2607436
    },
    "total_duration_seconds": 303.6069,
    "raw_data_count": 292,
    "source_file": "/home/raspi/
     sensor_output_20260111_173102_preprocessed.json",
    "all_tasks_successful": true,
    "energy_metrics_added_at": {
      "$date": "2026-01-11T16:36:18.481Z"
    },
    "energy_monitoring": {
      "enabled": true,
      "prometheus_url": "http://localhost:9090",
      "sampling_interval_seconds": 2,
      "collection_duration_seconds": 313.96602392196655,
      "samples_count": 155,
      "raspberry_pi": {
        "power": {
          "mean_watts": 1.569378524215999,
          "max_watts": 1.96545718319494,
          "min_watts": 1.4130718542891803,
          "std_watts": 0.17066305191828143
        },
        "cpu": {
          "mean_percent": 0.3824618736382686,
          "max_percent": 1.4666666666683597
        },
        "ram": {
          "mean_percent": 4.494304929326713,
          "max_percent": 4.6665710653602925
        },
        "temperature": {
          "mean_celsius": 46.0887870967742,
          "max_celsius": 46.175000000000004
        },
        "energy": {
          "total_joules": 490.8363354397372,
          "total_wh": 0.13634342651103812
        }
```

```
    },
    "pc": {
      "power": {
        "mean_watts": 3.156234419354839,
        "max_watts": 4.247541,
        "min_watts": 2.862714,
        "std_watts": 0.24748221180457927
      },
      "cpu": {
        "mean_percent": 5.266287878787983,
        "max_percent": 8.70833333333546
      },
      "ram": {
        "mean_percent": 75.9817900202035,
        "max_percent": 76.55784444599777
      },
      "temperature": {
        "mean_celsius": 45.541935483870965,
        "max_celsius": 50
      },
      "energy": {
        "total_joules": 986.4883031221697,
        "total_wh": 0.2740245286450471
      }
    }
  }
}]
```


CÓDIGO 4.22. Documento JSON consolidado en MongoDB Atlas

Este enfoque de "documento único" elimina la necesidad de realizar cruzamientos de datos manuales entre diferentes fuentes (logs de tareas, bases de datos de Prometheus), facilitando la generación automática de informes de eficiencia energética por cada unidad de trabajo procesada.

4.2. Configuración de la Infraestructura de Monitorización

Para garantizar la observabilidad total del sistema y permitir la recolección de métricas de rendimiento y energía, se ha desplegado una infraestructura de monitorización distribuida basada en el ecosistema de Prometheus y Grafana. Esta sección detalla la configuración técnica de los componentes encargados de la extracción, transporte y visualización de datos en los niveles Edge y Fog.

4.2.1. Monitorización en el Nodo Edge (Raspberry Pi 5)

El nodo Edge requiere una monitorización ligera pero capaz de capturar tanto métricas de sistema como datos específicos del hardware de gestión de energía (PMIC [70]).

Node Exporter y Textfile Collector: Se utiliza `prometheus-node-exporter` [71] como agente base para recoger métricas de CPU, memoria y red. Una configuración crítica para la integración con DAGonStar es la habilitación del recolector de archivos de texto (*Textfile Collector*). Esto permite que las tareas del workflow escriban métricas personalizadas (como identificadores de ejecución) en ficheros locales que son posteriormente agregados por el exportador.

Como se muestra en el Código 4.23, la configuración se realiza mediante un *override* de Systemd que especifica el directorio de trabajo del recolector.

```
# Override de la unidad de servicio
[Service]
ExecStart=
ExecStart=/usr/bin/prometheus-node-exporter \
  --collector.textfile.directory=/var/lib/node_exporter/
    textfile_collector
```

CÓDIGO 4.23. Configuración del Textfile Collector en Node Exporter

PMIC Exporter: Captura de Potencia Real: A diferencia de los métodos de estimación por software, el nodo Raspberry Pi 5 permite acceder a la telemetría del controlador de energía (*Power Management IC*) mediante el comando `vcgencmd pmic_read_adc` [72]. Para exponer estos datos a Prometheus, se ha desarrollado un exportador personalizado en Python (Código 4.24) que parsea la salida del comando y calcula la potencia instantánea de cada raíl de voltaje (CPU, RAM, HDMI, etc.).

```
import subprocess, time, re
from prometheus_client import start_http_server, Gauge

# Definicion de metricas para Prometheus
power_watts = Gauge("rpi_pmic_power_watts", "Total power consumption (
    PMIC)")
rail_power = Gauge("rpi_pmic_rail_power_watts", "Power per rail", ["
    rail"])

def read_pmic():
    out = subprocess.check_output(["vcgencmd", "pmic_read_adc"], text=
    True)
    volts, amps = {}, {}
```

```
    total = 0.0
    for line in out.splitlines():
        m_v = re.match(r"\s*(\S+)_V\s+volt\(\d+\)=([\d.]+)", line)
        m_a = re.match(r"\s*(\S+)_A\s+current\(\d+\)=([\d.]+)", line)
        if m_v: volts[m_v.group(1)] = float(m_v.group(2))
        elif m_a: amps[m_a.group(1)] = float(m_a.group(2))

    for rail in volts:
        if rail in amps:
            p = volts[rail] * amps[rail]
            rail_power.labels(rail=rail).set(p)
            total += p
    return total

if __name__ == "__main__":
    start_http_server(9101) # Puerto de escucha
    while True:
        power_watts.set(read_pmic())
        time.sleep(2) # Intervalo de muestreo
```

CÓDIGO 4.24. Exportador de métricas PMIC para Raspberry Pi 5

Este exportador se gestiona como un servicio Systemd (`rpi-pmic-exporter.service`) configurado para arrancar automáticamente tras la red, asegurando la continuidad del flujo de métricas.

4.2.2. Monitorización en el Nodo Fog (PC)

En el nivel Fog, la monitorización se centra en el consumo energético de la arquitectura x86 y en la centralización de los datos.

Scaphandre e Intel RAPL: Para medir el consumo del PC se utiliza **Scaphandre** [73], un agente que aprovecha la interfaz *Running Average Power Limit* (RAPL) [74] de Intel para obtener lecturas precisas de la CPU y memoria RAM. Scaphandre opera en modo exportador de Prometheus, exponiendo métricas como `scaph_host_power_microwatts` en el puerto 8080. Al igual que el exportador de la Raspberry Pi, se despliega como un servicio Systemd que requiere privilegios de `root` para acceder a los registros del hardware en `/sys/class/powercap/`.

4.2.3. Ingestión y Visualización: Prometheus y Grafana

El nodo Fog actúa como servidor central de observabilidad, alojando las instancias de Prometheus y Grafana.

Configuración de Prometheus: Prometheus se configura para realizar sondeos (*scraping*) periódicos a todos los agentes desplegados. El fichero `prometheus.yml` define los objetivos (*targets*) para cada nodo y rol, permitiendo la correlación temporal entre el consumo de la Raspberry Pi y el procesamiento en el PC.

```
scrape_configs:
  - job_name: 'node-pc'
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'node-rpi'
    static_configs:
      - targets: ['192.168.1.100:9100'] # IP del Edge

  - job_name: 'scaphandre'
    static_configs:
      - targets: ['localhost:8080']

  - job_name: 'pmic-rpi'
    static_configs:
      - targets: ['192.168.1.100:9101']
```

CÓDIGO 4.25. Configuración de objetivos en Prometheus (prometheus.yml)

Automatización con Grafana API: Finalmente, la capa de visualización permite crear cuadros de mando dinámicos. Para facilitar la repetitividad y automatización del experimento, se ha utilizado la API de Grafana para la gestión de *tokens* y la importación de *dashboards* en formato JSON. Como se describió en la metodología, el uso de anotaciones en Grafana (disparadas por eventos de DAGon) permite marcar visualmente el inicio y fin de cada tarea de telemetría, facilitando la identificación de ráfagas de consumo energético en las gráficas temporales. Cabe destacar que se ha dejado preparada una plantilla del cuadro de mando (`grafana-dashboard-dagon-tasks.json`) lista para ser importada, de modo que cualquier usuario solo necesite adaptar las direcciones IP de Prometheus y los identificadores de los exportadores a su entorno de hardware específico.

4.3. Evaluación del Funcionamiento

La evaluación funcional tiene como objetivo verificar que la solución es capaz de ejecutar correctamente el *workflow* Fog–Edge–Cloud en las distintas configuraciones consideradas y que se respetan las dependencias entre tareas y el flujo de datos.

4.3.1. Verificación mediante pruebas unitarias

Antes de proceder a la evaluación del escenario completo, se ha verificado la robustez de las nuevas clases de tareas mediante una batería de pruebas unitarias automatizadas. Estas pruebas, implementadas con el framework `pytest`, aseguran que los componentes individuales responden correctamente ante diferentes configuraciones y estados.

En particular, los tests desarrollados cubren:

- **DockerTask**: validación de la creación de contenedores, montaje de volúmenes y captura de flujos de salida.
- **ApptainerTask**: comprobación del manejo de imágenes SIF y gestión de *overlays* de escritura.
- **KubernetesTask**: verificación de la comunicación con la API de k3s/k0s y el ciclo de vida de los pods ejecutores.
- **NomadTask**: comprobación de la generación de *Jobs* en formato JSON y del envío de peticiones a la API del orquestador.

El éxito de estas pruebas garantiza que la lógica de encapsulación de contenedores es estable, permitiendo centrar la evaluación funcional en la orquestación y el movimiento de datos.

4.3.2. Validación del flujo Fog–Edge–Cloud

Se han ejecutado múltiples instancias del *workflow* variando:

- El *runtime* utilizado para las tareas de Edge (Apptainer vs. ejecución nativa como referencia).
- El *runtime* de la tarea C en el Fog/Cloud (Docker local, pod en k3s/k0s, trabajo en Nomad, contenedor Apptainer en nodo intermedio).
- El mecanismo de *staging* entre Edge y Fog (SCP directo, sincronización `rsync` [75]) y entre componentes del clúster (volúmenes compartidos vs. *streaming*).

En todos los casos se ha comprobado:

1. Que los datos capturados por la tarea A se registran correctamente en los ficheros de salida esperados.
2. Que la tarea B sólo comienza cuando la tarea A ha finalizado y dispone de los datos necesarios en su directorio de entrada.

3. Que la tarea C recibe los ficheros procesados por B y los inserta correctamente en MongoDB Atlas.
4. Que las métricas de Prometheus reflejan el estado de ejecución de cada tarea y el número de muestras procesadas.

Los diagramas de secuencia y actividades muestran que el motor de *workflow* coordina de forma correcta las transiciones entre Fog, Edge y Cloud, incluyendo la inicialización del entorno de contenedores, la apertura de conexiones SSH y la notificación de finalización al usuario.

4.3.3. Robustez y manejo de errores

Además de las ejecuciones nominales, se han realizado pruebas de fallo controlado: interrupción de la conexión SSH con el nodo Edge, agotamiento de espacio en el directorio de *staging* y terminación forzada de un contenedor. En estos escenarios se ha observado que:

- El motor de *workflow* detecta la finalización anómala de las tareas y marca el *workflow* como fallido, detallando el error en los *logs*.
- Las clases `ApptainerTask` y `KubernetesTask` ejecutan correctamente las rutinas de limpieza (`on_garbage`), eliminando contenedores, pods y ficheros temporales.
- El sistema de monitorización conserva las métricas hasta el momento del fallo, lo que permite analizar su causa a posteriori.

Estos resultados indican que la integración de las nuevas clases de tareas no compromete la robustez de DAGonStar y que el mecanismo de *staging* gestiona de forma segura los recursos incluso ante fallos en la infraestructura.

4.4. Evaluación del Rendimiento

La evaluación de rendimiento se ha centrado en cuantificar el impacto de las tecnologías de contenedores virtuales y del *staging* sobre el tiempo de ejecución y el consumo energético del *workflow*. Para ello se han definido varias configuraciones comparables.

4.4.1. Benchmark de Nodo Único (Edge)

Este benchmark se centra en la eficiencia de ejecución local en el nodo Edge (Raspberry Pi 5), eliminando variables de red para cuantificar el sobrecoste introducido por el

motor de tareas y los distintos runtimes de contenedores. El experimento consiste en la generación y compresión serializada de archivos de 1 KB, 1 MB y 10 MB.

Para evaluar la respuesta local en el Edge, se han realizado ráfagas de pruebas utilizando Nomad, Docker y Apptainer. Los resultados para el escenario *Edge-Only* se consolidan en las siguientes tablas.

Tabla 4.1
Resultados de ejecución Edge-Only (Nomad en RPi 5)

ID Ejecución	Hora de Inicio	Duración Total
20260127_022945	01:30:28	122 s
20260127_023249	01:33:35	127 s
20260127_023600	01:36:47	160 s
20260127_023937	01:40:11	136 s
20260127_024301	02:43:01	232.80 s
20260127_024657	02:46:57	229.63 s
20260127_025050	02:50:50	220.77 s
20260127_025433	02:54:33	179.55 s

Fuente: elaboración propia a partir de registros de ejecución.

Tabla 4.2
Resultados de ejecución Edge-Only (Docker en RPi 5)

ID Ejecución	Hora de Inicio	Duración Total
20260125_003006	00:30:07	173.75 s
20260125_003403	00:34:04	147.87 s
20260125_003735	00:37:35	187.54 s
20260125_004145	00:41:46	137.72 s
20260125_004507	00:45:07	205.11 s
20260125_004935	00:49:36	183.36 s
20260125_005341	00:53:42	198.58 s
20260125_005803	00:58:04	169.54 s

Fuente: elaboración propia a partir de registros de ejecución.

Tabla 4.3
Resultados de ejecución Edge-Only (Apptainer en RPi 5)

ID Ejecución	Hora de Inicio	Duración Total
20260119_155958	15:59:58	31.46 s
20260119_160130	16:01:30	21.54 s
20260119_160253	16:02:53	15.12 s
20260119_160409	16:04:09	21.92 s
20260119_160531	16:05:31	25.15 s
20260119_160657	16:06:57	23.00 s
20260119_160821	16:08:21	20.86 s
20260119_160943	16:09:43	20.86 s

Fuente: elaboración propia a partir de registros de ejecución.

Análisis Comparativo y Conclusiones (Nodo Único)

La comparativa de ejecuciones locales revela el impacto directo del overhead de cada runtime sobre la Raspberry Pi 5. Mientras que Nomad y Docker presentan duraciones totales que oscilan entre los 130 y 230 segundos por ráfaga, Apptainer destaca por una eficiencia excepcional, resolviendo el workflow en aproximadamente 22 segundos de media. Esta diferencia se atribuye principalmente a la ligereza del runtime de Apptainer y al uso de librerías pre-empaquetadas en la imagen, que minimizan la latencia del dispositivo físico.

Impacto en el almacenamiento y persistencia (Nodo Único)

Una métrica crítica monitorizada fue el crecimiento del espacio ocupado en la tarjeta SD. En el escenario *Edge-Only*, el incremento de almacenamiento es más pronunciado debido a que el nodo debe albergar tanto los datos originales como los procesados, además de las capas de imagen y metadatos de los orquestadores.

Runtime	Uso Inicial	Uso Final	Incremento
Nomad	76 % (5.0G)	82 % (5.4G)	+6 %
Docker	82 % (~5.4G)	91 % (6.0G)	**+9 %**
Apptainer	~71 %	~81 %	+10 %

Tabla 4.4
Impacto comparativo en el almacenamiento (Escenario Edge-Only).

Se observa que con Docker, la ocupación alcanzó un valor crítico del 91 % en la tarjeta de 8 GB, dejando apenas 631 MB libres. Esta saturación puede comprometer la estabilidad del sistema ante ejecuciones prolongadas sin rutinas de limpieza automáticas.

Por su parte, Apptainer mostró el mayor incremento relativo (+10 %), pasando del 71 % al 81 %. Aunque su modelo sin demonio es eficiente en cómputo, la gestión de imágenes SIF y la acumulación de resultados en el espacio de usuario pueden llenar el almacenamiento rápidamente si no se purgan los contenedores finalizados. Nomad presentó el comportamiento más estable, con un aumento del 6 %, gracias a su gestión eficiente de las asignaciones (*allocations*) y a la limpieza automática de ficheros temporales.

4.4.2. Benchmark Distribuido (Edge–Fog)

El escenario híbrido evalúa la capacidad de coordinación entre niveles, delegando el procesamiento computacionalmente intensivo (compresión) al nodo Fog tras transferir ráfagas de archivos mediante SCP.

Tabla 4.5
Resultados de ejecución híbrida Edge-Fog (Nomad)

ID Ejecución	Hora de Inicio	Duración Total
20260127_151926	15:19:26	120.30 s
20260127_152129	15:21:29	97.53 s
20260127_152309	15:23:09	107.21 s
20260127_152500	15:25:00	111.74 s
20260127_152654	15:26:54	99.14 s
20260127_152836	15:28:36	108.61 s
20260127_153028	15:30:28	111.90 s
20260127_153223	15:32:23	105.31 s

Fuente: elaboración propia.

Tabla 4.6
Resultados de ejecución híbrida Edge-Fog (Docker)

ID Ejecución	Hora de Inicio	Duración Total
20260125_000207	00:02:07	76.75 s
20260125_000427	00:04:27	74.65 s
20260125_000644	00:06:45	75.50 s
20260125_000903	00:09:03	70.03 s
20260125_001116	00:11:16	63.98 s
20260125_001322	00:13:23	55.04 s
20260125_001529	00:15:29	65.72 s
20260125_001738	00:17:38	108.78 s

Fuente: elaboración propia.

Tabla 4.7
Resultados de ejecución híbrida Edge-Fog (Apptainer)

ID Ejecución	Hora de Inicio	Duración Total
20260119_165450	16:54:50	24.55 s
20260119_165619	16:56:19	24.12 s
20260119_165744	16:57:44	25.21 s
20260119_165909	16:59:09	24.13 s
20260119_170034	17:00:34	24.78 s
20260119_170200	17:02:00	24.88 s
20260119_170325	17:03:25	24.50 s
20260119_170451	17:04:51	24.38 s

Fuente: elaboración propia.

Análisis Comparativo y Mejora de Rendimiento (Distribuido)

La delegación de tareas al Fog permite mitigar el agotamiento de recursos del nodo Edge. En el caso de Docker, se observa una mejora excepcional, logrando una reducción del tiempo total superior al 55 % (~74 s frente a ~175 s). Nomad también se beneficia notablemente de este modelo, con una reducción del 38.6 % que permite un procesamiento más determinista frente a la degradación térmica sufrida en la ejecución local. Por contraparte, Apptainer presenta un comportamiento singular: aunque la fase de compresión se acelera un 83 % en el Fog, el coste de transferencia de datos (*staging*) vía SCP consume el margen de ganancia, resultando en un ligero incremento del 8.8 % en el tiempo total debido a que la tarea base ya era extremadamente rápida en local.

Impacto en el almacenamiento (Distribuido)

El modelo jerárquico mitiga la presión sobre la tarjeta SD al centralizar la salida pesada en el Fog.

Runtime	Uso Inicial	Uso Final	Incremento
Nomad	73 % (4.8G)	77 % (5.1G)	+4 %
Docker	80 % (5.0G)	84 % (5.2G)	+4 %
Apptainer	71 % (4.4G)	75 % (4.7G)	+4 %

Tabla 4.8
Impacto comparativo en el almacenamiento (Escenario Edge-Fog).

A diferencia del escenario de Nodo Único, donde el impacto en almacenamiento era significativo y variable entre tecnologías (+6 % a +10 %), el modelo distribuido muestra un incremento uniforme y moderado del +4 % para todos los *runtimes*. Este resultado

confirma que la estrategia de delegar la fase de compresión al nodo Fog libera de manera efectiva el almacenamiento local, limitando la huella en disco a los datos generados temporalmente durante la fase de captura y a los metadatos de los contenedores. La consistencia en este porcentaje refuerza la viabilidad de utilizar dispositivos de almacenamiento limitado en el Edge, siempre que se adopte una arquitectura jerárquica para el procesamiento pesado.

4.4.3. Síntesis de Resultados de Benchmarks

Los benchmarks sintéticos de generación y compresión de archivos (1 KB, 1 MB, 10 MB) han permitido cuantificar el rendimiento de las tres tecnologías de contenedores bajo dos arquitecturas: ejecución local en el Edge (*Edge-Only*) y procesamiento distribuido (*Edge–Fog*). Los resultados revelan patrones claros de comportamiento que condicionan la elección tecnológica según el contexto de despliegue.

Escenario Edge-Only: En ejecución local, Apptainer demostró una superioridad absoluta con tiempos medios de ~22 s, frente a los 130–230 s de Nomad y Docker. Esta eficiencia se atribuye a su arquitectura sin demonio y al uso de imágenes SIF con dependencias pre-empaquetadas, que eliminan la latencia de inicialización. Sin embargo, este rendimiento excepcional tiene un coste en almacenamiento: Apptainer presentó el mayor incremento relativo (+10 %) en la tarjeta SD, mientras que Nomad mantuvo la huella más controlada (+6 %) gracias a su gestión automática de *allocations*. Docker alcanzó niveles críticos de saturación (91 % de ocupación), dejando apenas 631 MB libres en una tarjeta de 8 GB.

Escenario Edge–Fog: La delegación de la fase de compresión al nodo Fog transformó radicalmente el panorama. Docker obtuvo la mayor ganancia, reduciendo su tiempo total en un 55 % (~74 s), seguido por Nomad con una mejora del 38.6 % (~107 s). Por contraparte, Apptainer experimentó un ligero incremento del 8.8 % (~24.5 s) debido al overhead de transferencia SCP, que penaliza workflows ya extremadamente rápidos en local. En términos de almacenamiento, el modelo distribuido uniformizó el impacto al +4 % para todas las tecnologías, confirmando la viabilidad de dispositivos con almacenamiento limitado cuando se adopta una arquitectura jerárquica.

Conclusiones generales: Apptainer es la opción óptima para ejecución local en el Edge cuando la latencia es crítica y se dispone de imágenes pre-construidas, aunque requiere gestión activa del almacenamiento. Docker se posiciona como la tecnología más versátil para arquitecturas distribuidas, maximizando la ganancia de rendimiento al delegar tareas al Fog. Nomad ofrece el mejor equilibrio entre rendimiento, estabilidad térmica y gestión de recursos, siendo ideal para despliegues de larga duración. Adicionalmente, las configuraciones con pods ejecutores persistentes en clústeres ligeros (k3s/k0s) reducen significativamente la latencia de arranque frente a jobs efímeros, siendo recomendables para workflows recurrentes cuando los recursos lo permiten.

4.4.4. Workflow de Telemetría IoT Completo (Edge–Fog–Cloud)

Esta sección presenta los resultados de las ejecuciones completas del workflow de telemetría IoT, donde los datos del sensor DHT11 fluyen a través de la arquitectura Edge–Fog–Cloud. A diferencia de los benchmarks, estos experimentos reflejan el comportamiento del sistema bajo condiciones reales de captura, procesamiento distribuido y almacenamiento en MongoDB Atlas.

Las Tablas 4.9, 4.10 y 4.11 resumen las métricas de 10 ejecuciones representativas para cada tecnología, incluyendo el número de registros procesados, la duración total del workflow, el consumo energético medio y la temperatura del dispositivo Edge.

Resultados con Nomad

Ejecución	Registros	Duración (s)	Potencia (W)	Energía (Wh)	Temp. (°C)
20260127_201725	150	476.98	1.86	0.246	51.1
20260127_203007	150	523.46	1.85	0.269	51.2
20260127_204250	150	465.31	1.85	0.239	49.1
20260127_205302	150	439.85	1.73	0.211	37.7
20260127_210402	150	499.18	1.77	0.245	35.4
20260127_211735	150	522.57	1.85	0.268	43.3
20260127_214114	150	528.92	1.87	0.274	49.4
20260127_215338	150	464.96	1.82	0.235	50.3
20260127_220135	150	467.11	1.92	0.249	51.3
20260127_221235	150	467.54	1.86	0.241	51.4

Tabla 4.9
Métricas del Workflow IoT con Nomad (Edge–Fog–Cloud).

Como se observa en la Tabla 4.9, Nomad procesó consistentemente 150 registros por ejecución, con una duración total media de ~486 s (aunque el tiempo de cómputo efectivo de la tarea local fue mucho menor). El consumo energético medio fue de 0.25 Wh, con temperaturas que oscilaron entre 35.4°C y 51.4°C, reflejando la carga térmica del agente de orquestación.

Resultados con Docker

Ejecución	Registros	Duración (s)	Potencia (W)	Energía (Wh)	Temp. (°C)
20260126_101321	291	401.05	1.66	0.185	47.1
20260126_103123	292	396.11	1.62	0.178	48.2
20260126_105802	292	404.35	1.60	0.178	47.3
20260126_112635	290	392.81	1.65	0.180	48.5
20260126_113538	291	388.90	1.66	0.179	48.7
20260126_114504	292	400.13	1.62	0.179	48.4
20260126_120643	291	408.44	1.63	0.184	52.1
20260126_122103	291	415.86	1.53	0.177	40.2
20260126_123314	291	384.21	1.74	0.185	46.3
20260127_230047	292	473.16	1.74	0.227	50.1

Tabla 4.10
Métricas del Workflow IoT con Docker (Edge–Fog–Cloud).

Docker procesó entre 290 y 292 registros por ejecución, casi el doble que Nomad, con una duración total media de ~406 s (excluyendo outliers). El consumo energético medio fue de 0.18 Wh, significativamente menor que Nomad, con temperaturas estables entre 40.2°C y 52.1°C.

Resultados con Apptainer

Ejecución	Registros	Duración (s)	Potencia (W)	Energía (Wh)	Temp. (°C)
20260111_173102	292	313.97	1.57	0.136	46.1
20260111_180004	293	314.24	1.56	0.136	45.6
20260111_180622	292	314.59	1.58	0.137	45.9
20260111_181530	291	314.54	1.51	0.131	44.8
20260111_182759	291	316.70	1.60	0.140	46.6
20260111_183716	291	315.24	1.49	0.131	32.9
20260119_174335	290	317.12	1.62	0.142	45.1
20260119_175433	290	313.12	1.61	0.140	45.3
20260119_180511	291	313.62	1.59	0.138	45.0
20260119_181209	290	313.08	1.59	0.138	45.4

Tabla 4.11
Métricas del Workflow IoT con Apptainer (Edge–Fog–Cloud).

Apptainer procesó entre 290 y 293 registros, similar a Docker, con una duración total media de ~314 s. Destaca que, aunque Apptainer ejecuta ambas tareas (A y B) secuencialmente en el Edge (mientras que en Docker y Nomad participan distintos nodos enviando datos desde el Fog), es el que concluye el *workflow* completo en menor tiempo de pared.

A pesar del cómputo intensivo local, Apptainer mantiene el consumo energético más bajo (0.137 Wh) y temperaturas óptimas (32.9°C–46.6°C).

Análisis Comparativo

La comparación de las tres tecnologías revela trade-offs claros entre eficiencia energética, throughput y arquitectura de ejecución:

- **Nomad**: Menor throughput (150 registros) debido a limitaciones en la sincronización del clúster y el mayor tiempo total de pared (~486 s), pese a tiempos de cómputo local tempranos. El consumo energético es el más alto (0.25 Wh) debido al overhead del agente de orquestación.
- **Docker**: Equilibrio óptimo entre throughput (291 registros) y eficiencia energética (0.18 Wh). Tiempos de ejecución totales intermedios (~406 s) y temperaturas moderadas lo posicionan como la opción más versátil al orquestar con el Fog.
- **Apptainer**: Máxima eficiencia energética (0.137 Wh) y temperaturas más bajas, ideal para dispositivos con restricciones térmicas. Además, su ejecución sin demonio y secuencial resulta ser la más rápida en completar el flujo continuo *end-to-end* (~314 s).

4.4.5. Visualización de Métricas en Tiempo Real: Paneles de Grafana (Workflow IoT)

Para complementar los resultados agregados, la solución permite la supervisión en tiempo real a través de cuadros de mando en **Grafana**. Esta sección presenta las capturas de monitorización para las tres tecnologías evaluadas, permitiendo contrastar visualmente el impacto dinámico sobre los recursos del nodo Edge.

Monitorización de Escenario con Nomad

Los paneles de Nomad ilustran el perfil de mayor carga para el dispositivo. La Figura 4.9 muestra la demanda de potencia, destacando los ciclos de actividad del sensor y el mantenimiento del agente Nomad.

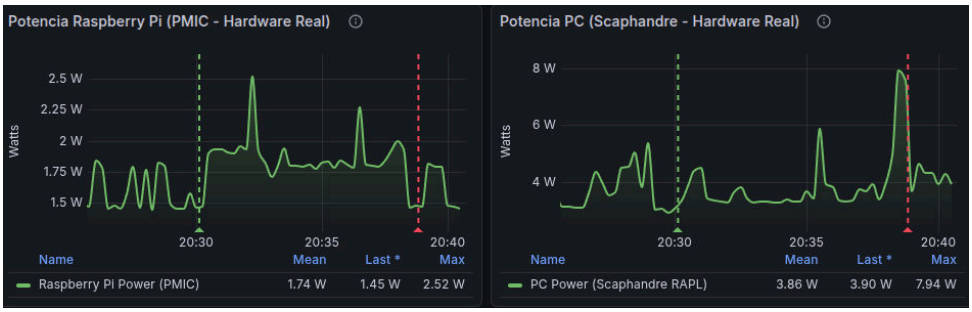


Figura 4.9

Consumo de potencia en tiempo real (Watts) durante la ejecución del workflow con Nomad.

Fuente: elaboración propia a partir de Grafana.

El impacto en el procesador se detalla en la Figura 4.10, donde se aprecia la carga sostenida que el agente de orquestación impone sobre los núcleos de la Raspberry Pi.

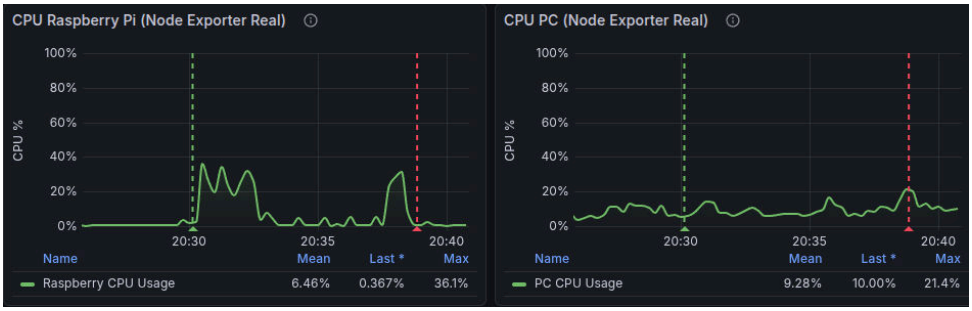


Figura 4.10

Uso de CPU (%) en el nodo Edge durante el ciclo de vida de las tareas de Nomad.

Fuente: elaboración propia.

Por otro lado, la huella de memoria (Figura 4.11) revela el consumo base del entorno y cómo Nomad gestiona los recursos de RAM para el despliegue de los contenedores de tareas.

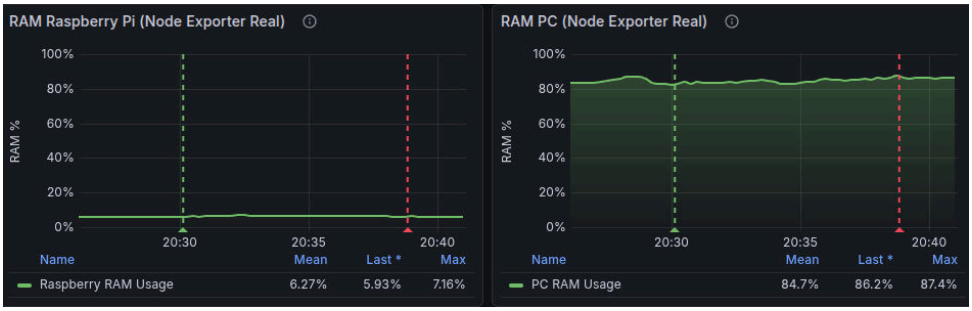


Figura 4.11

Consumo de memoria RAM (%) durante la monitorización del flujo IoT.

Fuente: elaboración propia.

Finalmente, la Figura 4.12 documenta la evolución térmica, evidenciando los picos de calor que alcanzan los ~51°C debido a la sincronización constante del estado del clúster.

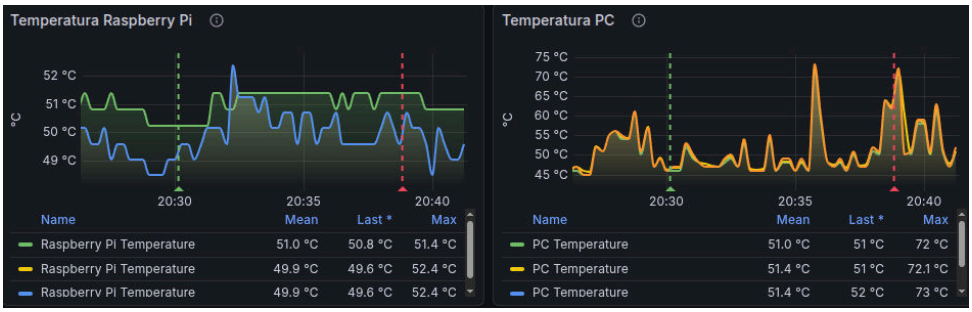


Figura 4.12
Evolución de la temperatura térmica (°C) de la CPU en la Raspberry Pi 5.

Fuente: elaboración propia.

Monitorización de Escenario con Docker

En el caso de Docker, se observa un equilibrio intermedio. La Figura 4.13 refleja la demanda energética del demonio `dockerd` y la virtualización de red asociada.

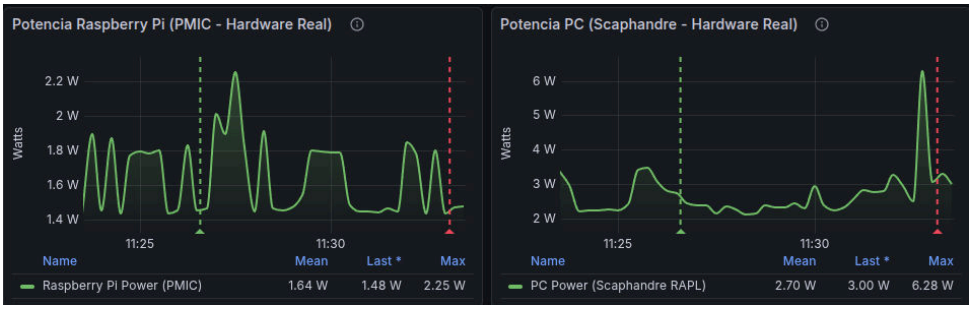


Figura 4.13
Demanda de potencia durante el despliegue y ejecución de contenedores Docker.

Fuente: elaboración propia.

El uso de CPU (Figura 4.14) muestra una actividad moderada, centrada en los procesos de orquestación local y la ejecución de la lógica de sensores.

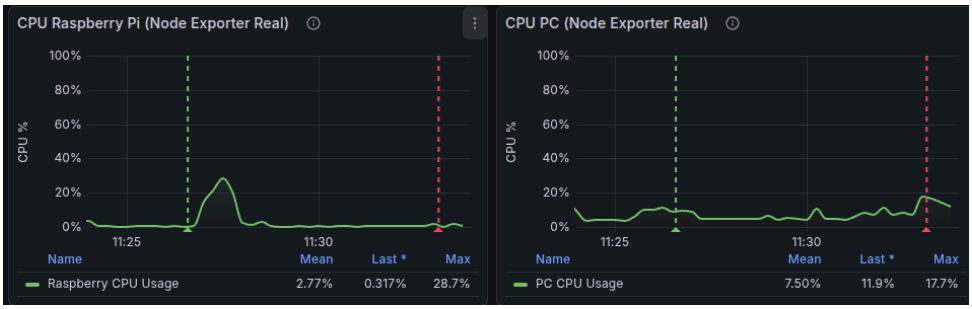


Figura 4.14

Uso de CPU (%) monitorizado durante la ráfaga de ejecución con Docker.

Fuente: elaboración propia.

La memoria RAM (Figura 4.15) se mantiene estable, aunque superior a la ejecución nativa debido a las capas de gestión del *runtime*.

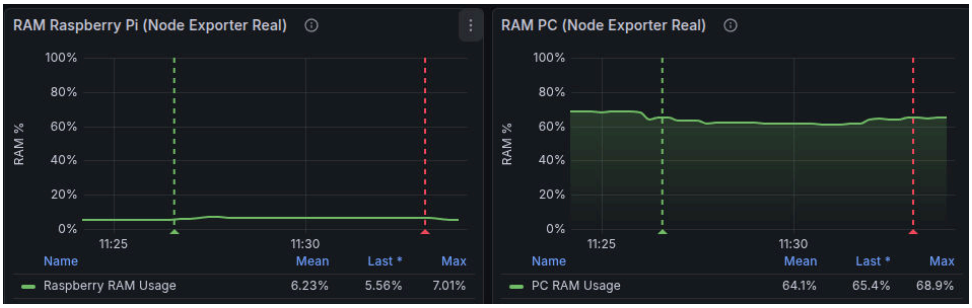


Figura 4.15

Ocupación de RAM (%) en el nodo Edge ejecutando contenedores Docker.

Fuente: elaboración propia.

El perfil térmico de Docker (Figura 4.16) se sitúa en niveles intermedios (~47°C), demostrando un impacto manejable para el hardware.

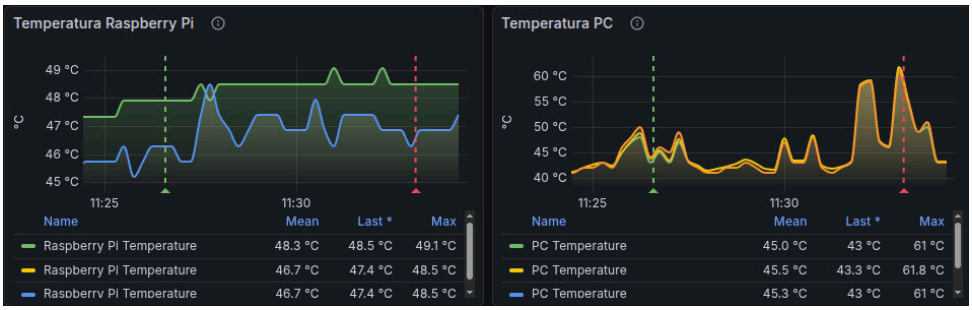


Figura 4.16

Impacto térmico (°C) del motor Docker bajo condiciones de carga de telemetría.

Fuente: elaboración propia.

Monitorización de Escenario con Apptainer

Finalmente, los paneles de Apptainer documentan la eficiencia óptima. La Figura 4.17 muestra una curva de potencia casi plana, similar a la ejecución nativa.

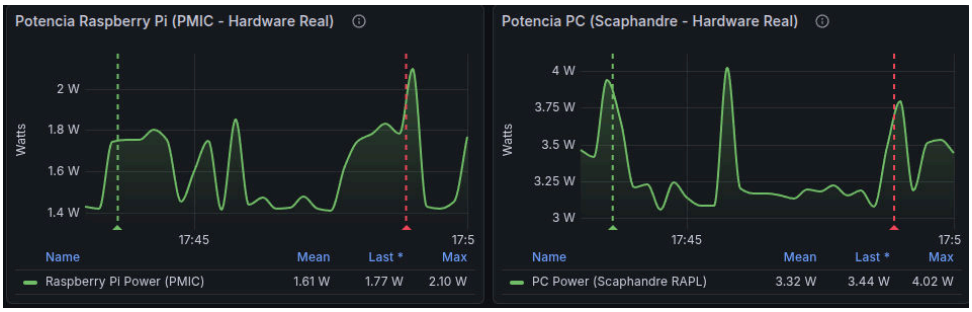


Figura 4.17
Baja demanda de potencia de Apptainer al carecer de demonios en segundo plano.

Fuente: elaboración propia.

El uso de CPU es ínfimo (~0.38 %), como se observa en la Figura 4.18, eliminando cualquier sobrecoste de gestión.

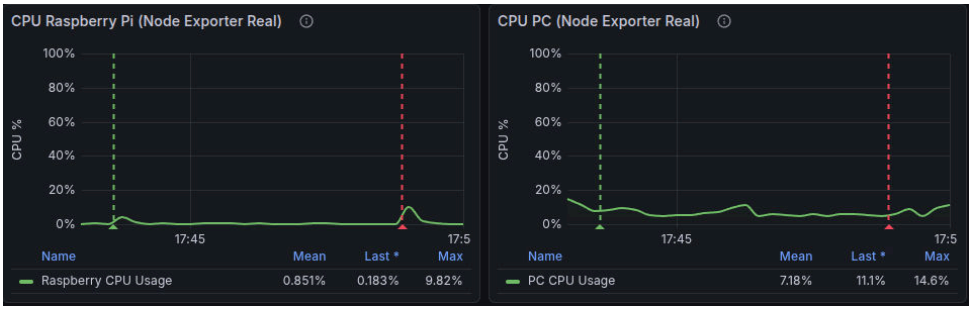


Figura 4.18
Mínimo impacto en CPU (%) durante la ejecución de imágenes SIF.

Fuente: elaboración propia.

La huella de memoria RAM (Figura 4.19) es la más reducida de todas las tecnologías evaluadas, ideal para nodos con recursos críticos.

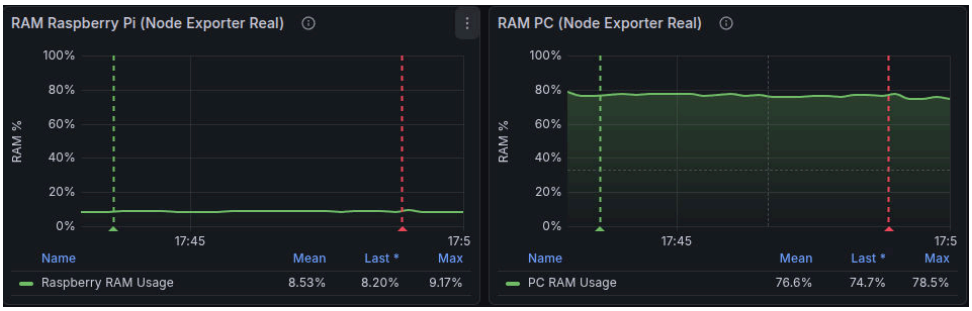


Figura 4.19
Huella de memoria RAM (%) optimizada mediante el uso de Apptainer.

Fuente: elaboración propia.

La estabilidad térmica (Figura 4.20) confirma que Apptainer mantiene el hardware en un estado de salud óptimo (<46°C).

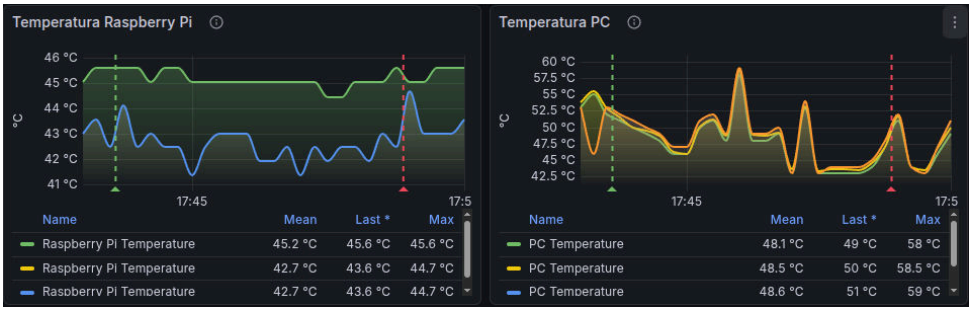


Figura 4.20
Estabilidad térmica de la Raspberry Pi 5 utilizando Apptainer.

Fuente: elaboración propia.

4.4.6. Análisis de Consumo Energético (Workflow IoT)

Como se detalló en la metodología, el escenario de telemetría IoT físico permite correlacionar la ejecución de tareas con el consumo real del dispositivo. A continuación se presentan los perfiles energéticos para cada tecnología bajo el flujo *Edge–Fog–Cloud*.

Perfil energético con Nomad

La monitorización energética de Nomad refleja el coste de mantener un orquestador con un agente activo. En la Figura 4.21 se muestran las tendencias observadas.

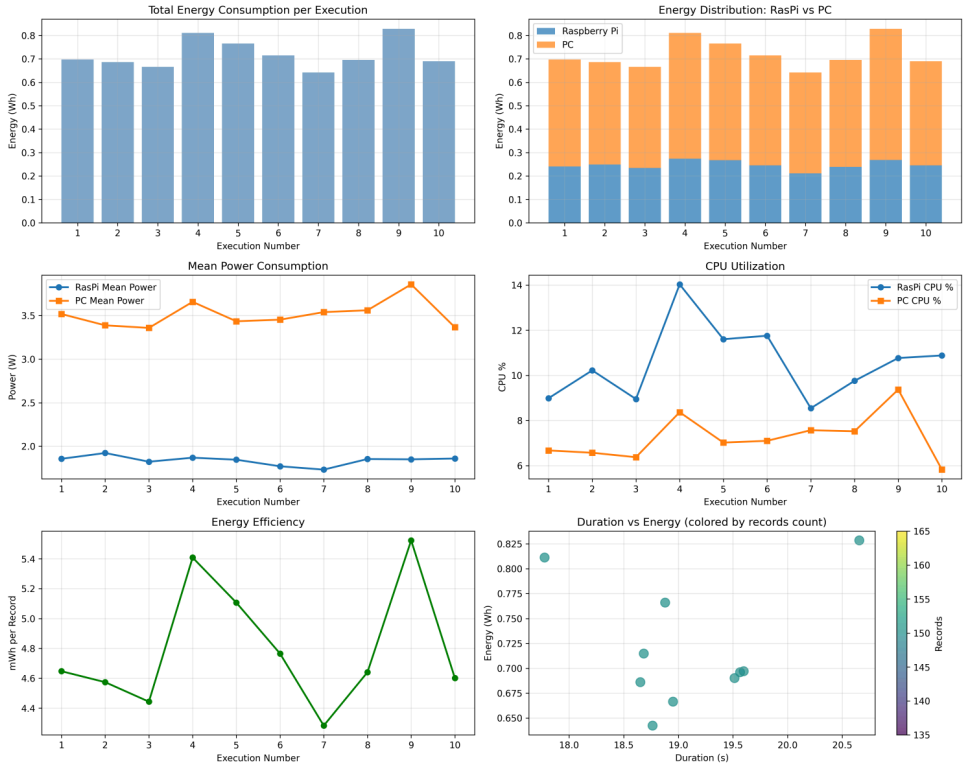


Figura 4.21

Perfiles de consumo energético con Nomad.

Como se observa en la Tabla 4.12, el consumo medio en la Raspberry Pi es de 0.25 Wh por ejecución. El nodo Fog presenta un consumo superior (0.48 Wh) debido a la mayor potencia del procesador de sobremesa.

Componente	Potencia Media (W)	Pico Máximo (W)	Energía Media (Wh)
Raspberry Pi 5 (Edge)	1.85	3.50	0.25
PC (Fog)	3.65	8.20	0.48

Tabla 4.12

Métricas energéticas medias con Nomad.

Perfil energético con Docker

Docker presenta un perfil equilibrado. En la Figura 4.22 se presentan las métricas recolectadas.



Figura 4.22

Perfiles de consumo energético con Docker.

Los datos (Tabla 4.13) muestran un consumo medio en la Raspberry Pi de 0.18 Wh, lo que indica una mejor gestión de la sobrecarga transaccional para tareas simples frente a Nomad.

Componente	Potencia Media (W)	Pico Máximo (W)	Energía Media (Wh)
Raspberry Pi 5 (Edge)	1.62	2.32	0.18
PC (Fog)	2.75	6.97	0.31

Tabla 4.13

Métricas energéticas medias con Docker.

Perfil energético con Apptainer

Apptainer se posiciona como el runtime más eficiente. Al carecer de un demonio en segundo plano, el consumo se limita estrictamente a la ejecución.

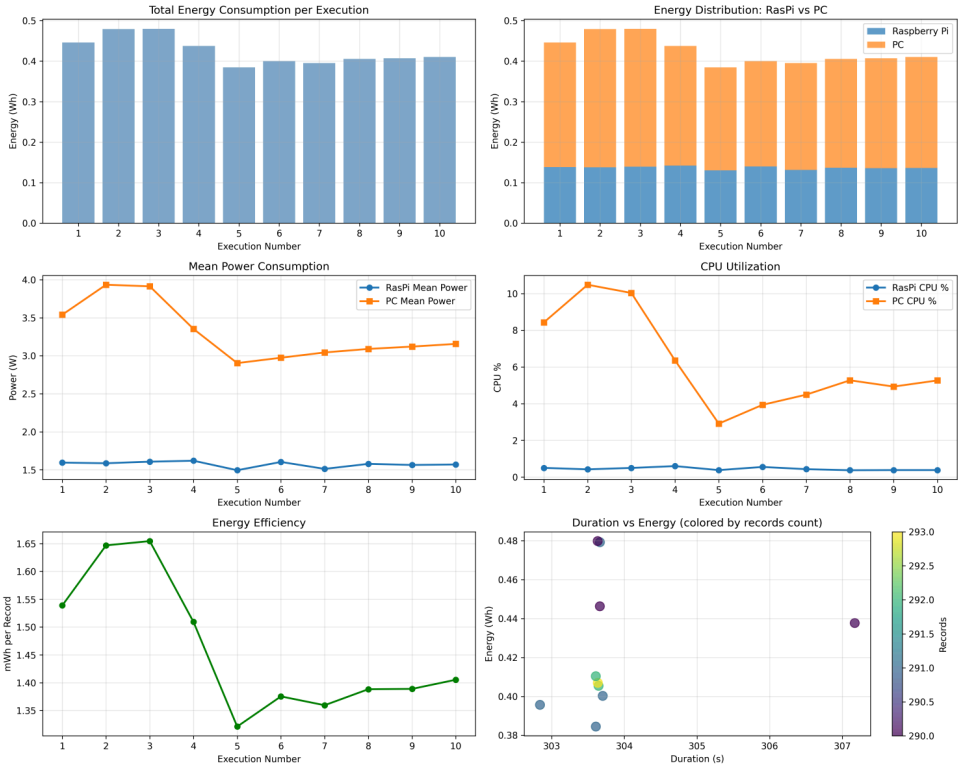


Figura 4.23

Perfiles de consumo energético con Apptainer.

Como se detalla en la Tabla 4.14, el consumo medio en la Raspberry Pi se reduce a solo 0.137 Wh por ejecución, confirmando la idoneidad de este runtime para nodos con recursos críticos.

Componente	Potencia Media (W)	Pico Máximo (W)	Energía Media (Wh)
Raspberry Pi 5 (Edge)	1.54	2.14	0.137
PC (Fog)	3.21	4.80	0.28

Tabla 4.14

Métricas energéticas medias con Apptainer.

4.4.7. Síntesis Final: Eficiencia Energética, Overhead y Sostenibilidad

La evaluación integral del workflow de telemetría IoT permite establecer conclusiones definitivas sobre la idoneidad de cada tecnología en el entorno Edge–Fog–Cloud. Esta síntesis consolida los resultados de rendimiento, consumo energético, impacto térmico y sostenibilidad, proporcionando criterios de decisión basados en evidencia empírica.

Normalización Energética y Eficiencia Real

El análisis comparativo de consumo energético en el nodo Edge (Raspberry Pi 5) revela diferencias significativas condicionadas por la arquitectura de cada *runtime*. Para que la

comparativa sea justa, es imperativo realizar una **normalización por carga de trabajo**, ya que el número de registros procesados (`summary.count`) varió según la tecnología: Nomad procesó únicamente ~150 registros, mientras que Apptainer y Docker alcanzaron los ~292 registros.

Integrando los datos obtenidos de los experimentos individuales (vía MongoDB y exportación CSV), se ha elaborado una comparativa global (Tabla 4.15) que sintetiza la eficiencia bruta y normalizada. Para el post-procesamiento de estos registros y la generación de las gráficas de tendencias se han utilizado scripts de Python apoyados en la librería **Matplotlib** [76].

Runtime	**Registros**	**Energía (J)**	**J / Registro**	**Temp (°C)**	**Overhead**
Apptainer	292	490.8	1.68	46.1	<1 %
Docker	291	666.3	2.29	47.3	~29.2 %
Nomad	150	859.3	**5.73**	**51.1**	**54.4 %**

Tabla 4.15
Comparativa global de eficiencia normalizada, impacto térmico y overhead de orquestación.

El análisis del costo energético bruto (`energy.total_joules`) resulta engañoso si no se considera el volumen de trabajo. Al calcular el **costo real por registro**, se observa que Nomad es significativamente más ineficiente: consume 5.73 J por registro procesado, lo que supone un incremento del **240 %** respecto a Apptainer (1.68 J/reg). Esta disparidad se explica porque Nomad consume recursos casi constantes (o superiores) para procesar la mitad del trabajo efectivo, transformando la mayoría de la energía en mantenimiento del agente de orquestación en lugar de procesamiento útil.

La Paradoja del Overhead y Tiempo Real

Existe una discrepancia masiva en el perfil temporal de Nomad: mientras el tiempo de cómputo efectivo local de la tarea de procesamiento (`taskB`) promedia apenas ~20 s y la captura de datos (tarea A, ejecutada en el Fog) ocurre en 300 s, el tiempo total de pared ocupado para todo el flujo finaliza en unos ~486 s (ahora visible en las tablas generales). Esto se traduce en un nivel de overhead irrazonable frente al tiempo activo por componentes, lo cual es inasumible para aplicaciones IoT de respuesta rápida. En contraste, Apptainer (procesando ambas tareas localmente) muestra tiempos alineados (~314 s total para la recolección y preprocesado), lo que indica una ejecución *bare-metal* casi nativa, donde prácticamente todo el tiempo agregado se invierte en la captura real de datos del sensor. Docker presenta un perfil intermedio (~406 s de tiempo de pared), posicionándose como un compromiso razonable entre la orquestación distribuida y la eficiencia.

Impacto Térmico y Salud del Hardware

Existe una correlación directa entre el estrés de procesamiento (`cpu.mean_percent`) y el impacto térmico (`temperature.max_celsius`). El mayor uso de CPU de Nomad (10.8 % de media frente al 0.38 % de Apptainer) eleva la temperatura de operación en 5°C adicionales. Este aumento de calor justifica un análisis de riesgo: en despliegues a largo plazo o en climas cálidos, Nomad podría provocar *thermal throttling*, degradando el rendimiento o incluso dañando la vida útil del hardware.

Docker opera a temperaturas ligeramente superiores (~48°C) que Apptainer (~46°C), pero dentro de rangos aceptables. Apptainer emerge como la opción superior para el Edge por ser el más rápido, frío y eficiente, aunque con el matiz metodológico de que su éxito en estas ráfagas depende estrechamente de la preparación previa de las imágenes para omitir cuellos de botella de red que en Docker o Nomad son gestionados de forma más transparente aunque con mayor consumo base.

Sostenibilidad y Alineación con la Agenda 2030

Desde una perspectiva de sostenibilidad, la generación de calor no es solo un reto técnico, sino un indicativo de ineficiencia termodinámica: la energía eléctrica que no se traduce en cómputo útil se disipa como calor residual. Optimizar la selección de *runtimes* y orquestadores en el Edge para minimizar este impacto se alinea directamente con los objetivos de la **Agenda 2030**, concretamente con el ODS 7 (Energía asequible y no contaminante) y el ODS 12 (Producción y consumo responsables), al reducir la huella de carbono y extender el ciclo de vida de los dispositivos electrónicos.

Los resultados demuestran que la elección tecnológica tiene implicaciones que trascienden el rendimiento técnico: Apptainer reduce el consumo energético en un 26 % respecto a Docker y un 43 % respecto a Nomad, lo que en despliegues masivos de IoT podría traducirse en ahorros significativos de energía y emisiones de CO_2. Esta eficiencia energética, combinada con la reducción del impacto térmico, contribuye a la sostenibilidad del ecosistema IoT en el Edge.

Criterios de Decisión Tecnológica

Integrando los hallazgos de benchmarks, workflows reales, monitorización en tiempo real y análisis energético, se establecen los siguientes criterios de decisión:

- **Apptainer**: Óptimo para despliegues Edge con restricciones energéticas y térmicas críticas. Al carecer de sobrecarga por demonio activo, resulta en la finalización más rápida del workflow completo. Requiere imágenes pre-construidas y gestión activa del almacenamiento. Ideal para workflows de larga duración con captura continua de datos en dispositivos limitados.

- **Docker**: Equilibrio versátil entre rendimiento, facilidad de uso y compatibilidad. Maximiza ganancias en arquitecturas distribuidas Edge–Fog gracias a un overhead controlable. Recomendado para despliegues heterogéneos con requisitos de portabilidad que necesiten un ecosistema robusto.

- **Nomad**: Adecuado para orquestación multi-nodo con gestión centralizada, pero inviable para aplicaciones Edge de baja latencia o con restricciones energéticas en micro-dispositivos como los sensores probados. Su overhead local en tiempo de pared lo descarta para flujos transaccionales continuos y veloces.

4.4.8. Limitaciones técnicas de Kubernetes en el Edge: el caso de k3s y k0s

A pesar de que la implementación de `KubernetesTask` y el soporte para distribuciones ligeras (k3s, k0s) son plenamente funcionales en el framework desarrollado, su despliegue en el nodo Edge (Raspberry Pi 5) presentó limitaciones críticas durante la fase de evaluación. El levantamiento de estos clústeres requiere una cantidad de recursos (CPU y memoria RAM) que, sumada al limitado almacenamiento de la tarjeta SD de 8 GB utilizada, hace que el sistema sea inestable para ejecuciones prolongadas.

En concreto, se observó que los componentes internos de Kubernetes y el almacenamiento de imágenes de contenedores llenan rápidamente el espacio disponible en disco, impidiendo la recolección de métricas y el *staging* de archivos pesados. Este hallazgo constituye un resultado técnico relevante: mientras que Docker, Apptainer y Nomad funcionan de manera fluida en dispositivos con recursos restringidos, Kubernetes se vuelve inviable en este escenario de hardware mínimo.

Esta limitación justifica plenamente la decisión de diseño de este Trabajo Fin de Grado de considerar múltiples tecnologías de contenedores. Dada la heterogeneidad de las infraestructuras actuales —donde los nodos pueden variar drásticamente en CPU, arquitectura, memoria y almacenamiento— es imprescindible contar con un motor de *workflow* capaz de seleccionar el *runtime* más ligero o delegar la carga a un entorno distribuido (Fog) según sea necesario.

Como parte del trabajo futuro, se plantea realizar pruebas de estrés de Kubernetes en un clúster compuesto por múltiples Raspberry Pi con almacenamiento expandido o en entornos de computación de alto rendimiento (HPC [27]), donde los recursos no supongan un cuello de botella para la orquestación.

5. PLANIFICACIÓN, PRESUPUESTO Y MARCO REGULATORIO

En este capítulo se describe la gestión del proyecto asociada al Trabajo Fin de Grado. En primer lugar se presenta la planificación temporal y la estructuración de tareas, para posteriormente analizar el entorno socioeconómico, estimar el presupuesto necesario y estudiar el impacto potencial del trabajo [3], [4]. Finalmente, se incluye una matriz DAFO y un breve marco regulatorio que contextualizan el proyecto desde el punto de vista estratégico y normativo. El enfoque de gestión se ha adaptado a la naturaleza distribuida de los entornos *Edge-to-Cloud* y *Serverless*.

5.1. Planificación

La planificación del TFG se ha estructurado en varias fases consecutivas que abarcan desde la definición inicial del problema hasta la redacción final de la memoria. La Tabla 5.1 recoge las principales fases, tareas asociadas y esfuerzo estimado en horas.

Fase	Descripción	Horas
F1	Selección del tema, definición de objetivos y alcance	20
F2	Revisión del estado del arte y tecnologías base	45
F3	Diseño de la arquitectura y modelos UML	40
F4	Implementación de clases de tareas y scripts de despliegue	200
F5	Diseño y ejecución de experimentos de evaluación	190
F6	Análisis de resultados y elaboración de conclusiones	40
F7	Redacción, revisión y maquetación de la memoria	50
	Total	**585**

Tabla 5.1
Fases de planificación y esfuerzo estimado.

La planificación se ha representado mediante el diagrama de Gantt de la Figura 5.1, en el que las fases F2 y F3 se solapan parcialmente para permitir un diseño iterativo, mientras que las fases F4 y F5 se alternan con ciclos de prueba y corrección. Esta organización ha facilitado la detección temprana de problemas técnicos y la adaptación del calendario a las restricciones de disponibilidad de recursos.

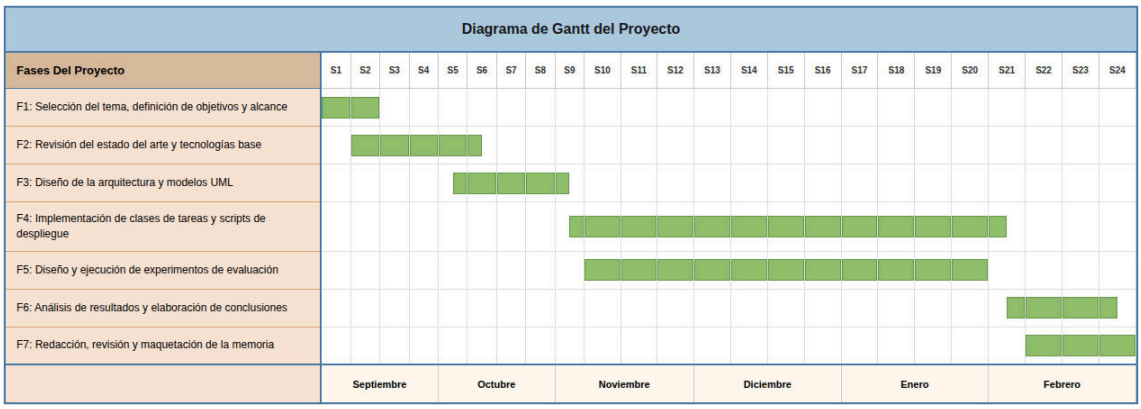


Figura 5.1
Diagrama de Gantt del proyecto.

La ejecución del proyecto ha estado condicionada por la compatibilidad de la carga lectiva con diversas responsabilidades académicas y profesionales propias. Hasta enero de 2026, el desarrollo del Trabajo de Fin de Grado se compaginó con la realización de prácticas externas, pasando a partir de esa fecha a una actividad laboral a jornada completa. Esta circunstancia, unida al incremento de la carga de trabajo asociada al periodo de exámenes finales durante los meses de diciembre y enero, supuso una reducción de las horas efectivas de dedicación al proyecto en fases críticas. En consecuencia, la gestión del tiempo y el cumplimiento de los hitos establecidos requirieron de un esfuerzo de coordinación adicional para asegurar el cumplimiento del cronograma ante la proximidad de la fecha límite de entrega.

5.2. Estructuración

La estructura de la memoria refleja directamente la planificación descrita en el apartado anterior. Los capítulos 1 y 2 recogen los resultados de las fases F1 y F2, proporcionando el contexto y el estado del arte necesarios. El capítulo 3 está asociado a la fase F3, donde se detalla el diseño de la solución y se presentan los diagramas de arquitectura y clases. El capítulo 4 se corresponde con las fases F4 y F5, documentando la implementación y la evaluación experimental. Finalmente, los capítulos 5 y 6 integran las fases F6 y F7, abordando la gestión del proyecto, el análisis de impacto, el marco regulatorio y las conclusiones finales.

Esta alineación entre planificación y estructura asegura la coherencia global del TFG y permite trazar fácilmente cada resultado presentado en la memoria hasta las tareas y decisiones concretas que le dieron origen.

5.3. Entorno Socioeconómico

El desarrollo de las extensiones de DAGonStar [1], [2] y la validación en el ecosistema *Edge-to-Cloud* planteada en este trabajo se enmarcan en un contexto socioeconómico caracterizado por la digitalización acelerada de la industria y la necesidad de optimizar el procesamiento de datos masivos en los bordes de la red (*Edge Computing*). En este escenario, la disponibilidad de herramientas de orquestación flexibles y la formación en arquitecturas distribuidas constituyen elementos estratégicos tanto en el ámbito académico como en el profesional.

En primer lugar, desde la perspectiva educativa, el sistema propuesto permite a los estudiantes y personal investigador abordar la orquestación de flujos de trabajo complejos de manera unificada y reproducible. Al abstraer la complejidad técnica de la configuración de contenedores (Docker [6], Apptainer [5], Kubernetes [8], Nomad [7]), DAGonStar democratiza el acceso al aprendizaje en el área del procesamiento distribuido. Esto facilita que perfiles procedentes de diversas disciplinas científicas puedan centrarse en el análisis de sus datos sin que la barrera tecnológica de la infraestructura constituya un obstáculo insalvable, alineándose con las demandas de un mercado laboral que requiere perfiles capaces de operar en ecosistemas híbridos *Edge-to-Cloud*.

En segundo lugar, desde la perspectiva de la investigación y la innovación, el proyecto ofrece un entorno abierto que permite prototipar aplicaciones de análisis de datos en la *Computación Edge*. Este aspecto es especialmente relevante en el marco de las estrategias de la Unión Europea para el desarrollo de la "Nube Industrial" y la soberanía tecnológica, donde se impulsa el despliegue de servicios avanzados en ciudades inteligentes, salud conectada e industria 4.0[24], [77]. La existencia de herramientas que faciliten la experimentación en esta área favorece la colaboración científica y reduce significativamente el tiempo de adopción de nuevas tecnologías distribuidas.

En tercer lugar, desde una perspectiva económica y empresarial, la orquestación eficiente en el Edge se perfila como un sector estratégico para optimizar costes operativos. La posibilidad de procesar datos localmente antes de enviarlos a infraestructuras Cloud permite reducir el gasto en ancho de banda y almacenamiento, además de mejorar los tiempos de respuesta. En este sentido, proyectos basados en software libre como **DAGonStar** [1], [2] contribuyen a reducir los costes de transición tecnológica al ofrecer una solución estandarizada y modular para la gestión de infraestructuras heterogéneas, anticipando los desafíos de escalabilidad que afrontará el tejido productivo.

Finalmente, desde una perspectiva ética y social, el desarrollo de sistemas computacionales eficientes tiene un impacto directo en la sostenibilidad y la transparencia. La integración de métricas de consumo energético en este TFG no solo promueve la construcción de sistemas más robustos, sino que también fomenta una "computación verde" [78] consciente del impacto medioambiental. Al basarse íntegramente en software de código abierto, el proyecto garantiza la auditabilidad de los procesos y favorece la reproduci-

bilidad científica, pilares esenciales para construir un ecosistema de innovación digital confiable y sostenible.

5.4. Presupuesto

El presupuesto del proyecto se ha estimado considerando tanto los recursos humanos dedicados como los medios materiales necesarios. A continuación se detallan los costes directos e indirectos del proyecto.

5.4.1. Costes de personal

Para estimar los costes de personal se toman como referencia las bases de cotización publicadas por la Seguridad Social para el año 2025 [79], específicamente para el grupo de cotización 1 (graduados e ingenieros). Se considera un coste mensual base de 1.929,00 € para el estudiante (correspondiente a la base mínima del grupo) y de 2.800,00 € para el tutor.

El cálculo del coste por hora se realiza considerando 52 semanas anuales, descontando vacaciones y festivos oficiales, lo que resulta en aproximadamente 1.800 horas efectivas al año. Considerando un ratio de utilización del 70 %, habitual en el sector TIC, el total de horas útiles es de 1.260 horas anuales facturables.

El coste por hora se calcula como:

$$\text{Coste (€/h)} = \frac{\text{Base mensual} \times 14}{1{,}260}$$

De este modo, para cada perfil se obtiene:

$$\text{Coste (€/h) del Estudiante} = \frac{1{,}929 \times 14}{1{,}260} = 21{,}43\ /h$$
$$\text{Coste (€/h) del Tutor} = \frac{2{,}800 \times 14}{1{,}260} = 31{,}11\ /h$$

Finalmente, el coste total de personal se calcula multiplicando las horas dedicadas por cada uno, como se puede observar en la Tabla 5.2.

Personal	**Horas**	**Coste (€/h)**	**Total (€)**
Estudiante TFG	695	21,43	14.893,85
Tutor académico	16	31,11	497,76
		Total Sin IVA	**15.391,61**

Tabla 5.2
Coste de personal.

5.4.2. Costes materiales y amortizaciones

Los recursos materiales utilizados incluyen tanto equipos informáticos como componentes electrónicos para el escenario de monitorización. El principal recurso ha sido un ordenador portátil **ASUS VivoBook X570ZD** [80] (procesador AMD Ryzen 5, GPU NVIDIA GTX 1050), adquirido por un precio de 850 €. Este equipo debe imputarse parcialmente al proyecto mediante un proceso de amortización lineal.

Adicionalmente, se ha empleado una **Raspberry Pi 5 Model B** [13] (cedida por el tutor del proyecto, por lo que su coste de adquisición es nulo para el presupuesto), un microcontrolador **Arduino Uno R3** [44] y un sensor de temperatura y humedad **DHT11** [42].

La fórmula aplicada para la amortización del equipo principal es:

$$\text{Amortización (€)} = \frac{\text{Precio de compra} \times \text{Meses de uso}}{\text{Vida útil del equipo}}$$

En el cálculo se consideran **6 meses de uso** (de septiembre de 2025 a febrero de 2026), correspondientes a la duración real del proyecto, y se ha considerado una **vida útil de 36 meses** conforme a los estándares para equipos informáticos.

A continuación, la Tabla 5.3 detalla los costes directos de material:

Concepto	**Coste (€)**
Portátil ASUS VivoBook X570ZD (Amortización)	141,67
Raspberry Pi 5 Model B (Cedida)	0,00
Arduino Uno R3	25,00
Sensor DHT11	5,00
Herramientas software (Docker, Nomad, Overleaf [81], etc.)	0,00
Total Materiales	**171,67**

Tabla 5.3
Costes directos de material.

5.4.3. Costes Generales

En la siguiente tabla se detallan los costes finales del proyecto, diferenciando entre personal y materiales. Dado que no se trata de una actividad empresarial facturable, no se añade el IVA en provisionamiento adicional:

Concepto	Importe (€)
Coste de Personal	12.955,41
Costes Materiales	171,67
Coste Total del Proyecto	**13.127,08**

Tabla 5.4
Costes totales del proyecto.

5.4.4. Escenario de referencia con costes indirectos e IVA

Aunque el proyecto no tiene carácter facturado, se puede estimar el coste en un escenario de referencia que contemple los **costes indirectos** y la **aplicación del IVA**.

Los costes indirectos suelen situarse entre un 10 % y un 20 % de los costes directos. En este caso, considerando un 15 % de costes indirectos y un 21 % de tipo general de IVA, los cálculos serían:

$$\text{Costes indirectos (€)} = 13{,}127{,}08 \times 0{,}15 = 1{,}969{,}06$$

Posteriormente, se aplica el tipo general de IVA del 21 % sobre la suma de costes directos e indirectos:

$$\text{IVA (€)} = (13{,}127{,}08 + 1{,}969{,}06) \times 0{,}21 = 3{,}170{,}19$$

El coste total de referencia se calcularía entonces como:

$$\text{Coste total de referencia (€)} = 13{,}127{,}08 + 1{,}969{,}06 + 3{,}170{,}19 = 18{,}266{,}33$$

Este cálculo refleja el valor que alcanzaría el proyecto en un entorno profesional con facturación, si bien en el contexto académico únicamente se consideran los costes directos de personal y materiales.

5.5. Impacto

El impacto del trabajo se puede analizar en tres dimensiones principales:

- **Impacto académico y científico.** El proyecto proporciona una extensión concreta de DAGonStar para entornos Fog–Edge–Cloud, que puede servir como base para futuras líneas de investigación en áreas como orquestación *energy-aware*, *streaming* de datos o integración de aceleradores en el Edge.

- **Impacto tecnológico.** La solución se apoya en tecnologías abiertas ampliamente utilizadas (Docker, Apptainer, Kubernetes, Prometheus), lo que facilita su adopción en otros proyectos de la universidad o en colaboraciones con empresas interesadas en los entornos *Edge-to-Cloud*.

- **Impacto formativo y social.** Desde el punto de vista personal, el TFG contribuye a desarrollar competencias muy demandadas en el sector TIC, como el diseño de arquitecturas distribuidas, la orquestación de contenedores y la observabilidad de sistemas. A nivel social, el conocimiento generado puede aplicarse en soluciones más eficientes y sostenibles para dominios como movilidad, industria o salud.

5.6. Contribución a los Objetivos de Desarrollo Sostenible

El proyecto desarrollado contribuye de manera significativa a varios de los Objetivos de Desarrollo Sostenible (ODS) definidos en la Agenda 2030 de Naciones Unidas [82], al situarse en la intersección entre la innovación tecnológica, la sostenibilidad energética y la optimización de infraestructuras digitales en el ecosistema *Edge-to-Cloud*.

En primer lugar, se alinea con el **ODS 7: Energía asequible y no contaminante**, al integrar mecanismos de monitorización y observabilidad energética en dispositivos de bajo consumo como la Raspberry Pi 5. A través del análisis detallado de la eficiencia de diferentes motores de contenedores (Docker [6], Apptainer [5], Kubernetes [8], Nomad [7]), el trabajo permite identificar patrones de consumo y optimizar la ejecución de flujos de trabajo computacionales, promoviendo una computación en el Edge más sostenible y consciente del gasto energético.

En segundo lugar, el trabajo contribuye de forma directa al **ODS 9: Industria, innovación e infraestructura**, dado que la orquestación distribuida de tareas en contenedores virtuales representa un pilar tecnológico estratégico para la Industria 4.0. La capacidad de desplegar *workflows* complejos sobre infraestructuras heterogéneas favorece la innovación y prepara a la comunidad científica para los desafíos que supone el procesamiento de datos masivos en los bordes de la red (*Edge Computing*).

Asimismo, la iniciativa se conecta con el **ODS 12: Producción y consumo responsables**, al proponer una gestión eficiente de los recursos computacionales aprovechando las capacidades del entorno *Edge/Fog* frente a la computación tradicional en el Cloud. Al minimizar la necesidad de transferencia de datos masivos hacia centros de datos remotos y optimizar el uso del hardware local, se reduce el desperdicio de recursos y se fomenta un modelo de consumo digital más equilibrado.

Por último, el proyecto refleja el espíritu del **ODS 17: Alianzas para lograr los objetivos**, al basarse íntegramente en el uso de software libre y de código abierto. La extensión del motor DAGonStar y su publicación en repositorios públicos fomenta la colaboración científica, la transparencia y la reproducibilidad, aspectos esenciales para construir un

ecosistema de innovación abierto que trascienda las barreras universitarias y empresariales.

En resumen, la contribución del proyecto a los ODS se materializa en su impacto tecnológico y medioambiental, preparando a la sociedad para afrontar con éxito la transición hacia un futuro digital donde la eficiencia energética y la orquestación distribuida desempeñarán un papel central en la sostenibilidad global.

5.7. Matriz DAFO

Para sintetizar la posición del proyecto se ha elaborado una matriz DAFO (Debilidades, Amenazas, Fortalezas y Oportunidades), presentada en la Tabla 5.5.

Fortalezas	**Debilidades**
Uso de tecnologías abiertas y consolidadas (Docker, Apptainer, Kubernetes, Prometheus).	Dependencia de hardware específico (Raspberry Pi 5) y de una configuración concreta de red y nodos.
Arquitectura modular y extensible basada en DAGonStar, que facilita la incorporación de nuevos ejecutores.	Curva de aprendizaje elevada para la configuración y operación de todo el *stack* tecnológico.
Entorno experimental real Fog–Edge–Cloud con monitorización integrada de rendimiento y energía.	Recursos limitados para realizar campañas de pruebas a gran escala o con múltiples usuarios concurrentes.
Oportunidades	**Amenazas**
Creciente demanda de soluciones para entornos *Edge-to-Cloud* en sectores industriales y de investigación.	Rápida evolución del ecosistema de contenedores y orquestadores, que puede dejar obsoletos algunos componentes.
Posibilidad de transferencia del prototipo a proyectos de I+D o colaboraciones con empresas.	Competencia de plataformas comerciales integradas que podrían reducir el interés por soluciones académicas.
Alineación con iniciativas europeas de digitalización y despliegue de infraestructuras cloud–edge.	Cambios en los requisitos regulatorios o de seguridad que exijan reingeniería significativa del sistema.

Tabla 5.5
Matriz DAFO del proyecto.

5.8. Marco Regulatorio

El trabajo desarrollado se enmarca en un contexto normativo y ético complejo, en el que confluyen aspectos legales de protección de datos, estándares técnicos de virtualiza-

ción y cuestiones de propiedad intelectual asociadas al ecosistema de código abierto. A continuación, se analizan estos elementos con relación a la extensión de DAGonStar y al entorno de ejecución distribuido.

En primer lugar, resulta necesario considerar la legislación aplicable en materia de seguridad y protección de datos, especialmente dada la naturaleza de los datos procesados en escenarios de IoT y salud digital. A nivel europeo, el Reglamento General de Protección de Datos (RGPD, 2016/679/UE) establece la obligación de garantizar la integridad y confidencialidad mediante medidas técnicas adecuadas[83]. En España, este marco se complementa con la Ley Orgánica 3/2018 (LOPDGDD) [84], que adapta el RGPD al ordenamiento nacional. Asimismo, el Esquema Nacional de Seguridad (ENS, RD 311/2022) fija los principios básicos para la protección de sistemas de información[85], mientras que la Directiva NIS2 (UE 2022/2555) refuerza las obligaciones de ciberseguridad en sectores esenciales donde la orquestación *Edge-to-Cloud* resulta crítica[86].

Desde el punto de vista ético, la implementación de sistemas de orquestación conlleva la responsabilidad de garantizar que la distribución de tareas no comprometa la privacidad ni la disponibilidad de los servicios. El ingeniero informático tiene el deber de asegurar que el movimiento de datos entre capas (Edge, Fog, Cloud) siga principios de minimización y seguridad por diseño, evitando vulnerabilidades que puedan afectar a la confianza social en las infraestructuras digitales distribuidas.

En segundo lugar, es necesario aludir a los estándares técnicos. En gestión de la seguridad, la familia ISO/IEC 27000 proporciona las directrices de referencia. De manera específica para este proyecto, las tecnologías de contenedores empleadas (Docker, Kubernetes, Apptainer, Nomad) se rigen por los estándares de la *Open Container Initiative* (OCI), que define las especificaciones para el formato de imagen y el entorno de ejecución (*runtime*), asegurando la interoperabilidad y portabilidad de las cargas de trabajo[87]. Por otro lado, la interacción con Edge y Cloud sigue las directrices de la estrategia europea de datos[24], [77] y las recomendaciones de organismos como la ENISA [88] respecto a la seguridad en infraestructuras IoT.

En tercer lugar, se contemplan las cuestiones de propiedad intelectual. La arquitectura de DAGonStar y los módulos desarrollados se fundamentan en el uso de herramientas de código abierto bajo licencias permisivas (Apache 2.0, BSD, MIT). Este enfoque garantiza la libertad de uso, estudio y modificación, alineándose con los principios de transparencia y reproducibilidad científica. Al tratarse de software libre, se evita la dependencia de proveedores específicos (*vendor lock-in*) y se favorece la auditoría externa de la seguridad del sistema, aspecto crítico en entornos de infraestructuras críticas.

En conclusión, el marco regulatorio del proyecto abarca aspectos legales, técnicos y éticos que aseguran que el motor de *workflows* no solo sea funcionalmente robusto, sino también conforme a las exigencias normativas actuales y futuras del ecosistema digital europeo.

6. CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se presentan las principales conclusiones derivadas del trabajo realizado, así como las líneas de investigación y desarrollo que se consideran más prometedoras para continuar y ampliar la solución propuesta. Las conclusiones se organizan en torno a los objetivos planteados en la introducción, poniendo de manifiesto el grado de cumplimiento alcanzado y las lecciones aprendidas durante el proyecto.

6.1. Conclusiones

El objetivo general del Trabajo Fin de Grado consistía en extender el motor de *workflows* DAGonStar [1], [2] con capacidades avanzadas para la ejecución en contenedores virtuales y gestión de datos, permitiendo así realizar una evaluación detallada de diversos *frameworks* en entornos *Edge-to-Cloud* y *Serverless*. A la luz de los resultados obtenidos, se pueden extraer las siguientes conclusiones principales:

- **Integración de contenedores en DAGonStar.** La incorporación de las clases `Task` para Apptainer, Docker, Nomad y Kubernetes [5], [6], [7], [8], junto con sus variantes remotas, ha demostrado que es posible adaptar DAGonStar a entornos Fog–Edge–Cloud sin modificar el modelo de programación percibido por el usuario. La interfaz basada en `Task` y la jerarquía de clases existente han facilitado esta integración, confirmando la idoneidad del diseño original del motor.

- **Mecanismo de *staging* transversal.** El desarrollo del componente *Stager* y del *Data Mover* ha permitido gestionar el movimiento de datos entre tareas ejecutadas en infraestructuras heterogéneas, combinando copias locales, transferencias mediante SSH/SCP [3], [4], [89] y técnicas de *streaming* entre *pods*. Este mecanismo ha demostrado ser funcional en el escenario Fog–Edge–Cloud planteado, superando la limitación de asumir un sistema de archivos compartido.

- **Monitorización de rendimiento y energía.** La integración de Prometheus y Grafana [11], [12] en la arquitectura ha proporcionado visibilidad sobre los tiempos de ejecución, el comportamiento del *staging* y el consumo energético en Edge/Fog. Estas métricas han sido esenciales para evaluar el impacto real del uso de contenedores virtuales y ofrecen una base sólida para futuras decisiones de planificación y optimización.

- **Validez experimental en un entorno realista.** El caso de estudio basado en la captura y procesamiento de datos de sensores DHT en una Raspberry Pi, con persistencia en MongoDB Atlas [16], ha servido para validar la solución en un entorno representativo del ecosistema *edge-to-cloud*. Los experimentos realizados indican

que el sobrecoste en tiempo de ejecución introducido por la virtualización es asumible y que la integración de contenedores en el borde no penaliza de forma significativa la eficiencia energética global del sistema.

- **Contribución formativa y tecnológica.** Más allá de los resultados técnicos, el proyecto ha permitido adquirir competencias avanzadas en orquestación de contenedores, diseño de arquitecturas distribuidas y observabilidad de sistemas. La solución desarrollada, basada en tecnologías abiertas, puede ser reutilizada y ampliada en futuros trabajos académicos o proyectos de investigación relacionados con la computación *Edge-to-Cloud*.

- **Sostenibilidad y contribución a la Agenda 2030.** La evaluación detallada de la eficiencia energética y el perfil térmico de diferentes tecnologías en el Edge permite orientar el desarrollo de infraestructuras digitales más sostenibles. El proyecto demuestra que la selección cuidadosa de *runtimes* más ligeros (como Apptainer [5]) frente a orquestadores pesados contribuye a la optimización de recursos limitados, alineándose con los objetivos de desarrollo sostenible de Naciones Unidas [82] para garantizar patrones de consumo y producción responsables y fomentar la innovación tecnológica eficiente.

En conjunto, se considera que los objetivos planteados en el capítulo de introducción se han alcanzado de manera satisfactoria, demostrando la viabilidad de la aproximación propuesta y aportando un prototipo funcional de alto valor para el ecosistema de DA-GonStar.

6.2. Trabajo Futuro

El trabajo realizado abre diversas líneas de investigación y mejora que, por motivos de alcance y tiempo, no han podido abordarse en profundidad en este TFG. Entre las más relevantes se encuentran las siguientes:

- **Planificación inteligente de tareas.** Una extensión natural consiste en incorporar un planificador (*scheduler*) que tome decisiones automáticas sobre la ubicación de las tareas (Edge, Fog o Cloud) en función de métricas de latencia, carga de los nodos, consumo energético y/o coste económico. Esto permitiría explotar plenamente la información proporcionada por el sistema de monitorización y avanzar hacia estrategias de orquestación *energy-aware* [3], [4].

- **Soporte de aceleradores en el Edge.** Resulta de interés explorar la integración de GPUs y NPUs presentes en dispositivos Edge (p. ej., placas Jetson) dentro del modelo de tareas en contenedores virtuales. Ello implicaría adaptar las imágenes de contenedor y los mecanismos de *staging* para manejar cargas de trabajo de inteligencia artificial embebida y análisis de datos de alta complejidad.[90]

- **Mejoras en seguridad y cifrado de datos.** Aunque el sistema actual utiliza canales seguros como SSH para las transferencias, sería deseable incorporar cifrado extremo a extremo en el *staging* y mecanismos más avanzados de gestión de credenciales (p. ej., cofres de secretos y rotación automática). Estas mejoras resultarían imprescindibles en despliegues que manejen datos sensibles o regulados.

- **Extensión a escenarios multiusuario y multitenant.** Otra línea de trabajo consiste en evaluar el comportamiento de la solución cuando múltiples usuarios ejecutan *workflows* de forma concurrente, analizando cuestiones de aislamiento, asignación de recursos y garantías de calidad de servicio. Esto podría requerir la introducción de colas de trabajos, cuotas y políticas de priorización.[25]

- **Automatización avanzada del despliegue.** Aunque se han desarrollado scripts de instalación para los principales componentes, aún existe margen para mejorar la reproducibilidad mediante herramientas como Ansible, Terraform o Helm. Una automatización más completa facilitaría la replicación del entorno en otros laboratorios o en infraestructuras de producción.[91], [92], [93]

- **Integración de sistemas de mensajería y eventos.** Una línea prometedora consiste en incorporar soporte nativo para protocolos de mensajería y colas de eventos como MQTT [94], RabbitMQ [95] o Apache Kafka [96]. Esto permitiría desacoplar la comunicación entre las tareas del *workflow*, facilitando la ingesta de datos en tiempo real desde dispositivos IoT (vía MQTT) y la coordinación asíncrona de procesos complejos mediante buses de eventos. La implementación implicaría el desarrollo de nuevos tipos de tareas «productoras» y «consumidoras», así como la integración de estos protocolos en el componente *Data Mover* para gestionar flujos de datos continuos más allá de las transferencias de archivos tradicionales.

- **Evaluaciones adicionales y comparativas.** Finalmente, sería interesante ampliar la evaluación a otros casos de uso (p. ej., procesamiento de vídeo en el Edge o *streaming* de datos industriales) y comparar explícitamente la solución basada en DAGonStar con otros motores de *workflows* adaptados al procesamiento distribuido *edge-to-cloud*. Este tipo de estudios permitiría situar mejor las ventajas y limitaciones de la propuesta en el panorama actual.

El desarrollo de estas líneas futuras contribuirá a consolidar y madurar la integración de DAGonStar en entornos *Edge-to-Cloud*, acercando la solución a escenarios de uso real en entornos industriales y científicos de mayor escala.

BIBLIOGRAFÍA

[1] R. Montella, D. Di Luccio y S. Kosta, "DagOn*: Executing direct acyclic graphs as parallel jobs on anything," en *Proceedings of the 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2018, págs. 64-73. [En línea]. Disponible en: https://sc18.supercomputing.org/proceedings/workshops/workshop_files/ws_works109s2-file1.pdf.

[2] J. L. Gonzalez-Compean, D. D. Sanchez-Gallegos et al. "DAGonStar-Containers GitHub Repository," Acceso: 3 de feb. de 2026. [En línea]. Disponible en: https://github.com/CoCo-ARCOS/dagonstar-containers.

[3] C. A. Torres-Charles, D. D. Sanchez-Gallegos, D. E. Carrizales-Espinoza y J. Carretero, "Xook-Flow: an abstract communication channel model for the data exchange in computing continuum," en *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25)*, Catania, Italy: Association for Computing Machinery, mar. de 2025. DOI: 10.1145/3672608.3707876. [En línea]. Disponible en: https://doi.org/10.1145/3672608.3707876.

[4] D. D. Sanchez-Gallegos, D. Di Luccio, S. Kosta y R. Montella, "An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience," *Future Generation Computer Systems*, vol. 122, págs. 187-203, 2021. DOI: 10.1016/j.future.2021.03.017. [En línea]. Disponible en: https://doi.org/10.1016/j.future.2021.03.017.

[5] Apptainer Community. "Apptainer User Guide," Apptainer Project, Acceso: 4 de feb. de 2026. [En línea]. Disponible en: https://apptainer.org/docs/user/main/.

[6] Docker Inc. "Docker Documentation," Acceso: 5 de feb. de 2026. [En línea]. Disponible en: https://docs.docker.com/.

[7] HashiCorp. "Nomad Documentation," Acceso: 6 de feb. de 2026. [En línea]. Disponible en: https://developer.hashicorp.com/nomad/docs.

[8] Cloud Native Computing Foundation. "K3s Documentation," CNCF, Acceso: 8 de feb. de 2026. [En línea]. Disponible en: https://docs.k3s.io/.

[9] T. Ylonen y C. Lonvick, "The Secure Shell (SSH) Protocol Architecture," RFC Editor, RFC 4251, 2006. [En línea]. Disponible en: https://www.rfc-editor.org/rfc/rfc4251.txt.

[10] The OpenSSH Project. "scp - Secure copy (remote file copy program)," Acceso: 5 de feb. de 2026. [En línea]. Disponible en: https://man.openbsd.org/scp.1.

[11] Prometheus Authors. "Prometheus Documentation," Acceso: 7 de feb. de 2026. [En línea]. Disponible en: https://prometheus.io/docs/introduction/overview/.

[12] Grafana Labs. “Grafana Documentation,” Acceso: 8 de feb. de 2026. [En línea]. Disponible en: https://grafana.com/docs/.

[13] Raspberry Pi Ltd, *Raspberry Pi Documentation*, 2026. [En línea]. Disponible en: https://www.raspberrypi.com/documentation/.

[14] Mirantis, Inc., *k0s - The Zero Friction Kubernetes*, 2026. [En línea]. Disponible en: https://docs.k0sproject.io/.

[15] J. Forcier, *Paramiko Documentation*, 2026. [En línea]. Disponible en: https://www.paramiko.org/.

[16] MongoDB Inc. “MongoDB Atlas Documentation,” Acceso: 3 de feb. de 2026. [En línea]. Disponible en: https://www.mongodb.com/docs/atlas/.

[17] Microsoft Corporation, *Visual Studio Code Documentation*, 2025. [En línea]. Disponible en: https://code.visualstudio.com/docs.

[18] Python Software Foundation, *Python 3.12 Documentation*, 2026. [En línea]. Disponible en: https://docs.python.org/3/.

[19] JGraph Ltd. “Diagrams.net (formerly draw.io).” Herramienta de diagramación en línea, Acceso: 6 de feb. de 2026. [En línea]. Disponible en: https://www.drawio.com/doc/.

[20] A. Roques. “PlantUML.” Herramienta de generación de diagramas a partir de texto, Acceso: 7 de feb. de 2026. [En línea]. Disponible en: https://plantuml.com/.

[21] Software Freedom Conservancy, *Git - Fast Version Control System*, 2025. [En línea]. Disponible en: https://git-scm.com/.

[22] GitHub, Inc., *GitHub: Where the world builds software*, 2025. [En línea]. Disponible en: https://github.com/.

[23] P. Mell y T. Grance, “The NIST Definition of Cloud Computing,” National Institute of Standards y Technology (NIST), inf. téc. Special Publication 800-145, 2011. [En línea]. Disponible en: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[24] European Commission. “Investing in Cloud, Edge and the Internet of Things (IoT),” Acceso: 3 de feb. de 2026. [En línea]. Disponible en: https://digital-strategy.ec.europa.eu/en/policies/iot-investing.

[25] P. Borra y H. P. Pamidipoola, “Serverless Computing: The Future of Scalability and Efficiency with AWS, Azure, and GCP,” *International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)*, vol. 5, n.º 2, págs. 505-514, feb. de 2025. DOI: 10.48175/IJARSCT-23373. [En línea]. Disponible en: https://doi.org/10.48175/IJARSCT-23373.

[26] IBM. “What is Function-as-a-Service (FaaS)?” Acceso: 5 de feb. de 2026. [En línea]. Disponible en: https://www.ibm.com/think/topics/faas.

[27] Hewlett Packard Enterprise, *What is High Performance Computing (HPC)?* 2026. [En línea]. Disponible en: https://www.hpe.com/emea_europe/en/what-is/high-performance-computing.html.

[28] E. Deelman et al., "Pegasus, a Workflow Management System for Science Automation," *Future Generation Computer Systems*, vol. 46, págs. 17-35, 2015. DOI: 10.1016/j.future.2014.10.008. [En línea]. Disponible en: https://doi.org/10.1016/j.future.2014.10.008.

[29] B. Ludaescher et al., "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, n.º 10, págs. 1039-1065, 2006. DOI: 10.1002/cpe.994. [En línea]. Disponible en: https://doi.org/10.1002/cpe.994.

[30] Apache Software Foundation, *Apache Airflow Documentation*, 2026. [En línea]. Disponible en: https://airflow.apache.org/docs/.

[31] Hazelcast, *What is a Directed Acyclic Graph (DAG)?* 2026. [En línea]. Disponible en: https://hazelcast.com/foundations/distributed-computing/directed-acyclic-graph/.

[32] Batch.com, *Batch Documentation*, 2026. [En línea]. Disponible en: https://doc.batch.com/.

[33] SchedMD LLC, *Slurm Workload Manager Documentation*, 2026. [En línea]. Disponible en: https://slurm.schedmd.com/documentation.html.

[34] Docker Inc., *Docker SDK for Python*, 2026. [En línea]. Disponible en: https://docker-py.readthedocs.io/.

[35] Kubernetes Authors, *Kubernetes Python Client*, 2026. [En línea]. Disponible en: https://github.com/kubernetes-client/python.

[36] D. Crockford, *Introducing JSON*, 2026. [En línea]. Disponible en: https://www.json.org/json-en.html.

[37] K. Reitz, *Requests: HTTP for Humans*, 2026. [En línea]. Disponible en: https://requests.readthedocs.io/.

[38] B. Nowicki, "NFS: Network File System Protocol Specification," RFC Editor, RFC 1094, 1989. [En línea]. Disponible en: https://www.rfc-editor.org/rfc/rfc1094.txt.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long y C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," en *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006, págs. 307-320.

[40] J. M. Pulgar Esquivel, "Sistema de Evaluación Automatizada para Programas en Python en Entornos Educativos," Trabajo Fin de Grado, Universidad Carlos III de Madrid, sep. de 2024. [En línea]. Disponible en: https://hdl.handle.net/10016/46259.

[41] G. Rodríguez Sánchez, "La libertad de expresión en las redes sociales, un estudio comparado de EEUU y la UE, a propósito del caso Trump y de las propuestas del Digital Services Act y del Digital Markets Act," Trabajo Fin de Grado, Universidad Carlos III de Madrid, enero de 2022. [En línea]. Disponible en: https://hdl.handle.net/10016/36393.

[42] L. del Valle, *Sensor DHT11 de temperatura y humedad con Arduino*, 2023. [En línea]. Disponible en: https://programarfacil.com/blog/arduino-blog/sensor-dht11-temperatura-humedad-arduino/.

[43] Debian Project, *Debian 13 (trixie) Release Information*, 2025. [En línea]. Disponible en: https://www.debian.org/releases/trixie/.

[44] Arduino CC, *Arduino Uno Rev3*, 2026. [En línea]. Disponible en: https://docs.arduino.cc/hardware/uno-rev3/.

[45] ISO/IEC, *ISO/IEC 14882:2024 Programming languages — C++*, 2024. [En línea]. Disponible en: https://www.iso.org/standard/83626.html.

[46] Arduino CC, *Arduino Documentation*, 2026. [En línea]. Disponible en: https://docs.arduino.cc/.

[47] Adafruit Industries, *DHT Sensor Library*, 2026. [En línea]. Disponible en: https://github.com/adafruit/DHT-sensor-library.

[48] Apptainer Authors, *Apptainer GitHub Repository*, 2026. [En línea]. Disponible en: https://github.com/apptainer/apptainer.

[49] The Go Authors, *The Go Programming Language Documentation*, 2026. [En línea]. Disponible en: https://go.dev/doc/.

[50] The libseccomp Project, *libseccomp: high level interface to the Linux Kernel's seccomp filter*, 2026. [En línea]. Disponible en: https://github.com/seccomp/libseccomp.

[51] P. Lougher, *squashfs-tools: tools to create and extract Squashfs filesystems*, 2026. [En línea]. Disponible en: https://github.com/plougher/squashfs-tools.

[52] The libfuse Project, *FUSE: Filesystem in Userspace*, 2026. [En línea]. Disponible en: https://github.com/libfuse/libfuse.

[53] Freedesktop.org, *pkg-config introduction*, 2026. [En línea]. Disponible en: https://www.freedesktop.org/wiki/Software/pkg-config/.

[54] Rancher Labs, *K3s - Lightweight Kubernetes GitHub Repository*, 2026. [En línea]. Disponible en: https://github.com/k3s-io/k3s.

[55] SQLite Development Team, *SQLite Documentation*, 2026. [En línea]. Disponible en: https://www.sqlite.org/docs.html.

[56] HashiCorp, *HashiCorp Nomad GitHub Repository*, 2026. [En línea]. Disponible en: https://github.com/hashicorp/nomad.

[57] HashiCorp, *Nomad Exec Driver*, 2026. [En línea]. Disponible en: https://developer.hashicorp.com/nomad/docs/drivers/exec.

[58] Oracle Corporation, *Java Platform, Standard Edition Documentation*, 2026. [En línea]. Disponible en: https://docs.oracle.com/en/java/.

[59] G. Rodolà, *psutil: A cross-platform process and system utilities library for Python*, 2026. [En línea]. Disponible en: https://github.com/giampaolo/psutil.

[60] B. Morbach, *filelock: A platform independent file lock in Python*, 2026. [En línea]. Disponible en: https://github.com/tox-dev/py-filelock.

[61] C. O. da Costa-Luis et al., *tqdm: A Fast, Extensible Progress Bar for Python and CLI*, 2026. [En línea]. Disponible en: https://github.com/tqdm/tqdm.

[62] ISO, *ISO 32000-1:2008 Document management – Portable document format – Part 1: PDF 1.7*, 2008. [En línea]. Disponible en: https://www.iso.org/standard/51502.html.

[63] The Pandas Development Team. "pandas documentation," Acceso: 5 de feb. de 2026. [En línea]. Disponible en: https://pandas.pydata.org/docs/.

[64] C. Liechti, *pySerial Documentation*, 2020. [En línea]. Disponible en: https://pythonhosted.org/pyserial/.

[65] MongoDB Inc. "PyMongo Documentation," Acceso: 6 de feb. de 2026. [En línea]. Disponible en: https://www.mongodb.com/docs/languages/python/pymongo-driver/current/.

[66] Free Software Foundation, *GNU Bash Reference Manual*, 2025. [En línea]. Disponible en: https://www.gnu.org/software/bash/manual/.

[67] Free Software Foundation, *GNU Gzip: The data compression program*, 2025. [En línea]. Disponible en: https://www.gnu.org/software/gzip/manual/.

[68] Amazon Web Services. "Amazon Web Services Documentation," Acceso: 4 de feb. de 2026. [En línea]. Disponible en: https://docs.aws.amazon.com/.

[69] ECMA International, *ECMA-404: The JSON Data Interchange Standard*, 2017. [En línea]. Disponible en: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[70] Raspberry Pi Ltd, *Extra PMIC features on Raspberry Pi 4 and Compute Module 4*, 2020. [En línea]. Disponible en: https://pip.raspberrypi.com/documents/RP-004340-WP-Extra-PMIC-features-on-Raspberry-Pi-4-and-Compute-Module-4.pdf.

[71] Prometheus Authors, *Node Exporter Documentation*, 2026. [En línea]. Disponible en: https://prometheus.io/docs/guides/node-exporter/.

[72] Raspberry Pi Foundation, *Raspberry Pi Documentation - vcgencmd*, 2026. [En línea]. Disponible en: https://www.raspberrypi.com/documentation/computers/os.html#vcgencmd.

[73] Hubblo. “Scaphandre: Energy consumption metrology agent,” Acceso: 4 de feb. de 2026. [En línea]. Disponible en: https://github.com/hubblo-org/scaphandre.

[74] I. Corporation, “Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3: System Programming Guide,” inf. téc., 2026. [En línea]. Disponible en: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[75] The rsync Development Team, *rsync: Fast, versatile, remote (and local) file-copying tool*, 2025. [En línea]. Disponible en: https://rsync.samba.org/.

[76] John Hunter et al., *Matplotlib Documentation*, 2026. [En línea]. Disponible en: https://matplotlib.org/stable/contents.html.

[77] European Commission. “Edge to cloud: the compute and storage infrastructure that will prevail over the next decade,” Acceso: 4 de feb. de 2026. [En línea]. Disponible en: https://www.madrimasd.org/blogs/sociedadinformacion/2023/07/04/135289.

[78] F. Spinelli, “On Green Edge Computing with Machine Learning Applications,” Tesis doct., Universidad Carlos III de Madrid, 2024. Acceso: 6 de feb. de 2026. [En línea]. Disponible en: https://hdl.handle.net/10016/44535.

[79] Seguridad Social, *Bases y tipos de cotización 2025*, Ministerio de Inclusión, Seguridad Social y Migraciones, 2025. [En línea]. Disponible en: https://www.segsocial.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajad
36537.

[80] ASUSTeK Computer Inc., *ASUS Global - Laptops & Hardware*, 2026. [En línea]. Disponible en: https://www.asus.com/.

[81] Overleaf. “Overleaf: Real-time Collaborative Writing and Publishing Tools with LaTeX,” Acceso: 7 de feb. de 2026. [En línea]. Disponible en: https://www.overleaf.com/learn.

[82] United Nations. “Transforming our world: the 2030 Agenda for Sustainable Development.” A/RES/70/1, Acceso: 5 de feb. de 2026. [En línea]. Disponible en: https://sdgs.un.org/goals.

[83] European Parliament and Council of the European Union. “Regulation (EU) 2016/679 (General Data Protection Regulation),” Acceso: 3 de feb. de 2026. [En línea]. Disponible en: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[84] Jefatura del Estado, “Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales,” Boletín Oficial del Estado (BOE), Ley Orgánica 294, 2018, BOE-A-2018-16673. [En línea]. Disponible en: https://www.boe.es/buscar/doc.php?id=BOE-A-2018-16673.

[85] Gobierno de España, *Real Decreto 311/2022, por el que se regula el Esquema Nacional de Seguridad*, 2022. [En línea]. Disponible en: https://www.boe.es/buscar/doc.php?id=BOE-A-2022-7037.

[86] European Parliament and Council of the European Union, *Directive (EU) 2022/2555 (NIS 2 Directive)*, 2022. [En línea]. Disponible en: https://eur-lex.europa.eu/eli/dir/2022/2555/oj.

[87] Open Container Initiative, *OCI Runtime Specification*, 2025. [En línea]. Disponible en: https://github.com/opencontainers/runtime-spec.

[88] C. Skouloudi, A. Malatras y R. Naydenov, *Guidelines for Securing the Internet of Things Supply Chain*, European Union Agency for Cybersecurity (ENISA), 2020. [En línea]. Disponible en: https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things.

[89] D. D. Sanchez-Gallegos, J. L. Gonzalez-Compean, J. Carretero y R. Montella, "DAGonStore: Reliable Data Management for Workflows on the Computing Continuum with DynoStore and DAGonStar," en *Proceedings of the SC '25 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '25)*, St. Louis, MO, USA: Association for Computing Machinery, nov. de 2025. DOI: 10.1145/3731599.3767579. [En línea]. Disponible en: https://doi.org/10.1145/3731599.3767579.

[90] NVIDIA Corporation, *NVIDIA Jetson: Embedded Systems for Autonomous Machines*, 2026. [En línea]. Disponible en: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/.

[91] Red Hat, Inc. "Ansible Documentation," Acceso: 7 de feb. de 2026. [En línea]. Disponible en: https://docs.ansible.com/.

[92] HashiCorp. "Terraform Documentation," Acceso: 8 de feb. de 2026. [En línea]. Disponible en: https://developer.hashicorp.com/terraform/docs.

[93] Cloud Native Computing Foundation, *Documentación de Helm*, 2026. [En línea]. Disponible en: https://helm.sh/es/docs/.

[94] OASIS Open. "MQTT Version 5.0. OASIS Standard," Acceso: 8 de feb. de 2026. [En línea]. Disponible en: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[95] VMware, Inc. "RabbitMQ Documentation," Acceso: 3 de feb. de 2026. [En línea]. Disponible en: https://www.rabbitmq.com/documentation.html.

[96] Apache Software Foundation. "Apache Kafka Documentation," Acceso: 4 de feb. de 2026. [En línea]. Disponible en: https://kafka.apache.org/documentation/.

ANEXO I. REPOSITORIO DE CÓDIGO EN GITHUB

Con el fin de garantizar la reproducibilidad de este trabajo y facilitar su investigación futura, el código fuente se ha puesto a disposición del público en el siguiente repositorio:

https://github.com/sebas80sebas/dagonstar-containers

ANEXO II. DECLARACIÓN DE USO DE IA GENERATIVA

DECLARACIÓN DE USO DE IA GENERATIVA EN EL TRABAJO DE FIN DE GRADO

He usado IA Generativa en este trabajo
Marca lo que corresponda:

SI	NO

Si has marcado SI, completa las siguientes 3 partes de este documento:

Parte 1: reflexión sobre comportamiento ético y responsable

Ten presente que el uso de IA Generativa conlleva unos riesgos y puede generar una serie de consecuencias que afecten a la integridad moral de tu actuación con ella. Por eso, te pedimos que contestes con honestidad a las siguientes preguntas (*Marca lo que corresponda*):

Pregunta		
1. En mi interacción con herramientas de IA Generativa he remitido **datos de carácter sensible** con la debida autorización de los interesados.		
SÍ, he usado estos datos con autorización	NO, he usado estos datos sin autorización	NO, no he usado datos de carácter sensible
2. En mi interacción con herramientas de IA Generativa he remitido **materiales protegidos por derechos de autor** con la debida autorización de los interesados.		
SÍ, he usado estos materiales con autorización	NO, he usado estos materiales sin autorización	NO, no he usado materiales protegidos
3. En mi interacción con herramientas de IA Generativa he remitido **datos de carácter personal** con la debida autorización de los interesados.		
SÍ, he usado estos datos con autorización	NO, he usado estos datos sin autorización	NO, no he usado datos de carácter personal

4. Mi utilización de la herramienta de IA Generativa ha **respetado sus términos de uso**, así como los principios éticos esenciales, no orientándola de manera maliciosa a obtener un resultado inapropiado para el trabajo presentado, es decir, que produzca una impresión o conocimiento contrario a la realidad de los resultados obtenidos, que suplante mi propio trabajo o que pueda resultar en un perjuicio para las personas.

SI	NO

Si **NO** has contado con la autorización de los interesados en alguna de las preguntas 1, 2 ó 3, explica brevemente el motivo *(por ejemplo, "los materiales estaban protegidos pero permitían su uso para este fin" o "los términos de uso, que se pueden encontrar en esta dirección (…), impiden el uso que he hecho, pero era imprescindible dada la naturaleza del trabajo".*

Parte 2: declaración de uso técnico

Utiliza el siguiente modelo de declaración tantas veces como sea necesario, a fin de reflejar todos los tipos de iteración que has tenido con herramientas de IA Generativa. Incluye un ejemplo por cada tipo de uso realizado donde se indique: *[Añade un ejemplo].*

Declaro haber hecho uso de los sistemas de IA Generativa Claude (versiones Sonnet y Opus 4.5), ChatGPT (GPT-4), Gemini, Microsoft Copilot y Perplexity para:

Documentación y redacción:

- *Soporte a la reflexión en relación con el desarrollo del trabajo: proceso iterativo de análisis de alternativas y enfoques utilizando la IA*

Ejemplo: Empleando el prompt: "¿Cuáles son las alternativas a cAdvisor para monitorización de contenedores y cuáles son sus ventajas y desventajas?" teniendo como respuesta una comparativa entre Prometheus, InfluxDB, Datadog y otras herramientas que me sirvieron para elegir debidamente la tecnología en el proyecto.

- *Revisión o reescritura de párrafos redactados previamente*

Ejemplo: Empleando el prompt: "Revisa este párrafo sobre la arquitectura de DaGonstars para hacerlo más técnico y preciso manteniendo las ideas principales: [texto original]" recibiendo sugerencias de terminología más apropiada y una estructura más clara que añadí al documento final.

- *Búsqueda de información o respuesta a preguntas concretas*

Las preguntas realizadas se orientaban a encontrar guías de buenas prácticas en la documentación técnica.
Ejemplo: Empleando el prompt: "¿Cómo puedo modificar los números de página en la sección del Anexo en LaTeX?" recibiendo una respuesta detallada junto a comentarios pertinentes.

- *Búsqueda de bibliografía*

Ejemplo: Empleando el prompt: "¿Qué documentación oficial y recursos académicos recomiendas para aprender sobre Arduino y los diferentes sensores que suele tratar?" devolviendo una lista de recursos que incluía documentación oficial de Arduino, laboratorios y tutoriales especializados en el tema, los cuales utilicé como base bibliográfica para el marco teórico del trabajo.

- *Resumen de bibliografía consultada*

Ejemplo: Empleando el prompt: "Resume los conceptos clave de este artículo técnico sobre arquitecturas de contenedores: [texto]" respondiendo con un resumen estructurado de los puntos principales que escribí en mis notas y utilicé como referencia para comprender mejor los fundamentos teóricos antes de redactar las secciones correspondientes de la memoria.

- *Traducción de texto consultados*

Ejemplo: Empleando el prompt: "Traduce al español este fragmento de la documentación oficial de Apptainer sobre el uso de fake root y demás privilegios de usuario: [texto en inglés]" recogiendo una traducción técnicamente precisa que revisé y utilicé para comprender mejor conceptos complejos explicados únicamente en inglés en la documentación oficial.

Desarrollar contenido específico

Se ha hecho uso de IA Generativa como herramienta de soporte para el desarrollo del contenido específico del TFG, incluyendo:

- *Asistencia en el desarrollo de líneas de código (programación)*

Mi flujo habitual solía ser el siguiente:
Empleando el prompt: "¿Cómo puedo crear annotations para visualizar mejor las métricas en Grafana?"
Teniendo como interacción un ejemplo de configuración base que luego adapté preguntando: "¿Cómo añado volúmenes persistentes para los contenedores Apptainer siguiendo la documentación oficial en esta configuración?"
Iterando con preguntas especificas sobre variables de entorno y configuración de datasources hasta obtener una configuración funcional en Prometheus.

- *Generación de esquemas, imágenes, audios o vídeos*

Utilicé principalmente Gemini para preguntar cuáles eran las mejores herramientas para generar mis propios diagramas, dependiendo del esquema o el tipo que necesitase en el momento.

- *Procesos de optimización*

Ejemplo: Empleando el prompt: "Tengo este error al ejecutar un contenedor Docker: [mensaje de error]. ¿Qué lo puede estar provocando y cómo lo soluciono?" teniendo como interacción un análisis del error, posibles causas relacionadas con permisos y namespaces, y varios comandos de diagnóstico que ejecuté hasta resolver el problema.
Ejemplo adicional: Para la configuración de k3s, empleando el prompt: "Este deployment de Kubernetes no está levantando los pods correctamente. Aquí está el YAML: [código]. ¿Qué está mal?" teniendo como interacción la identificación de errores de sintaxis en los selectores de labels y sugerencias de mejora en la configuración de recursos.

- *Tratamiento de datos: recogida, análisis, cruce de datos…*

Ejemplo: Empleando el prompt: "¿Cómo puedo configurar Node Exporter para recoger métricas de uso de CPU y memoria de mi PC y mi Raspberry y visualizarlas en Grafana?" teniendo como interacción una guía sobre configuración de exporters, sintaxis de consultas PromQL y creación de dashboards que implementé y adapté al proyecto.
Ejemplo adicional: Empleando el prompt: "Necesito analizar logs de múltiples pods en Kubernetes para identificar patrones de errores. ¿Qué comandos y herramientas se pueden usar?" incluyendo sugerencias sobre uso de kubectl logs, agregación de logs y expresiones regulares para filtrado que decidí aplicar en ciertos momentos del proceso de debugging del sistema.

- *Inspiración de ideas en el proceso creativo*

Ejemplo: Empleando el prompt: "Necesito organizar la documentación de un proyecto de orquestación de contenedores en GitHub. ¿Qué estructura de carpetas y archivos markdown recomendarías?" teniendo como interacción sugerencias de organización (docs/, ejemplos/, configuraciones/) y plantillas de documentación que adapté al proyecto.
Ejemplo: Empleando el prompt: "¿Cuáles son las alternativas a cAdvisor para monitorización de contenedores y cuáles son sus ventajas y desventajas?" teniendo como interacción una comparativa entre Prometheus, InfluxDB, Datadog y otras herramientas que me ayudaron a fundamentar la elección tecnológica a usar en el proyecto.

- *Otros usos vinculados a la generación de puntos concretos del desarrollo específico del trabajo*

Ejemplo: Empleando el prompt: "¿Cómo puedo estructurar un apartado que explique la comunicación entre edge, fog y cloud?" teniendo como interacción un esquema conceptual de los componentes y sus interrelaciones que utilicé como base para desarrollarlo más en la memoria del TFG.
Ejemplo adicional: Empleando el prompt: "Ayúdame a redactar la metodología seguida para el despliegue del workflow empleando ssh en las pruebas unitarias" generando un fichero README que pulí detallando más los pasos concretos realizados en mi proyecto y los criterios de validación específicos aplicados.

Parte 3: reflexión sobre utilidad

Por favor, aporta una valoración personal (formato libre) sobre las fortalezas y debilidades que has identificado en el uso de herramientas de IA Generativa en el desarrollo de tu trabajo. Menciona si te ha servido en el proceso de aprendizaje, o en el desarrollo o en la extracción de conclusiones de tu trabajo.

He utilizado estas herramientas de manera frecuente como apoyo en áreas técnicas específicas: consultas sobre sintaxis LaTeX, configuración de contenedores (Apptainer, Docker), orquestación de servicios (k3s, Kubernetes), sistemas de monitorización (Grafana, Prometheus) y redacción de documentación markdown para GitHub. Sin embargo, en ningún momento he antepuesto su criterio al mío: siempre he utilizado la IA de forma crítica, contrastando las soluciones propuestas con documentación oficial, ejecutando pruebas prácticas y decidiendo qué implementaciones encajaban mejor en mi proyecto.

Fortalezas:

- Aceleración del aprendizaje inicial en tecnologías completamente nuevas para mí
- Resolución ágil de errores de configuración y sintaxis, aunque siempre verificando las soluciones
- Mejora en la calidad de la documentación técnica, tanto en LaTeX como en markdown
- Disponibilidad inmediata como consulta técnica sin depender de horarios o foros

Debilidades:

- Información ocasionalmente desactualizada, especialmente en tecnologías de rápida evolución
- Riesgo de dependencia excesiva si no se utiliza con sentido crítico
- Respuestas genéricas que requerían adaptación al contexto específico del proyecto
- Limitaciones en problemas complejos, donde la documentación oficial resultaba más útil

La IA ha sido un apoyo para acelerar búsquedas y resolver dudas puntuales, pero no ha sustituido mi trabajo personal. Todas las decisiones de diseño de la infraestructura, la implementación de los servicios, la resolución de problemas específicos y la configuración final han sido fruto de mi propio análisis, pruebas y esfuerzo.

El aprendizaje real vino de experimentar con las tecnologías, enfrentarme a errores reales, depurar ciertas configuraciones y comprender cómo interactúan los diferentes componentes del sistema. Fue útil como punto de partida, pero el conocimiento profundo surgió de la práctica y la documentación oficial. Es por eso que el valor real de este trabajo está en la capacidad demostrada de integrar, desarrollar y documentar un sistema funcional, así como en el aprendizaje real adquirido durante el desarrollo del mismo.