

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

Cinvestav Unidad Tamaulipas

**Método de procesamiento
continuo para la construcción y
manejo de estructuras de
procesamiento agnósticas de la
infraestructura**

Tesis que presenta:

Dante Domizzi Sánchez Gallegos

Para obtener el grado de:

**Doctor en Ciencias
en Ingeniería y Tecnologías
Computacionales**

Director de la Tesis:
Dr. José Luis González Compeán

© Derechos reservados por
Dante Domizzi Sánchez Gallegos
2023

Este trabajo ha sido parcialmente apoyado por el proyecto 41756 “Plataforma tecnológica para la gestión, aseguramiento, intercambio y preservación de grandes volúmenes de datos en salud y construcción de un repositorio nacional de servicios de análisis de datos de salud”, financiado por FORDECYT-PRONACES’.

La tesis presentada por Dante Domizzi Sánchez Gallegos fue aprobada por:

Dr. Raúl Rivera Rodríguez

Dr. Rafael Mayo García

Dr. Ricardo Landa Becerra

Dr. Iván Lopez Arevalo

Dr. José Luis González Compeán, Director

Cd. Victoria, Tamaulipas, México., 11 de abril de 2023

A mi familia.

Agradecimientos

- A mis papás y hermanos por su apoyo durante esta etapa de mi vida.
- A mi director de tesis, el Dr. José Luis González Compeán por su dirección, experiencia, conocimiento, apoyo, tiempo, motivación y orientación durante mi estancia en el Cinvestav.
- A Diana por su apoyo y todos los momentos vividos juntos.
- A mis revisores, los Dres. Iván Lopez Arevalo, Ricardo Landa Becerra, Raúl Rivera Rodríguez y Rafael Mayo García por sus valiosos comentarios para mejorar este trabajo de investigación.
- Al Dr. Alejandro Galaviz y mis compañeros del CICESE-UFM por recibirme durante mi estancia en dicha institución.
- También agradezco al personal administrativo del CINVESTAV-Tamaulipas por su apoyo durante mi estadía en la unidad.
- Agradezco al CONACyT por el apoyo económico proporcionado, lo cual me ayudó realizar mis estudios de doctorado.

Índice General

Índice General	i
Índice de Figuras	v
Índice de Tablas	ix
Índice de Algoritmos	xii
Resumen	xiii
Abstract	xv
Nomenclatura	xvii
1. Introducción	1
1.1. Ciclo de vida de los productos digitales	2
1.2. Antecedentes y motivación	5
1.2.1. Estructuras de procesamiento para el manejo de CVPD	6
1.2.2. Modelos de despliegue de patrones de diseño para estructuras de procesamiento de datos	8
1.2.2.1. Modelos de programación	9
1.2.2.2. Modelos de comunicación	10
1.2.2.3. Modelos de coordinación de procesamiento	11
1.2.3. Independencia y autonomía de las estructuras de procesamiento	11
1.2.4. Dependencia de las estructuras de procesamiento con software, plataforma e infraestructuras	12
1.3. Planteamiento del problema	17
1.3.1. Premisas y formalización de conceptos	19
1.4. Hipótesis	20
1.5. Objetivo general y objetivos específicos	21
1.5.1. Objetivo general	21
1.5.2. Objetivos específicos	21
1.6. Panorama general del método para construcción de estructuras agnósticas	23
1.6.1. Agnosticismo en las estructuras de procesamiento	24
1.6.2. Metodología para crear estructuras de procesamiento agnósticas de la infraestructura	28
1.7. Metodología de investigación	31
1.8. Organización del documento	34

2. Marco teórico	35
2.1. Requerimientos de software funcionales y no funcionales	36
2.2. Arquitecturas de software	38
2.2.1. Arquitectura de N-capas	38
2.2.2. Arquitectura orientada al servicio	39
2.2.3. Arquitectura de microservicios	39
2.3. Patrones de diseño	40
2.4. Motores de flujos de trabajo	42
2.4.1. Paralelismo en flujos de trabajo	42
2.4.1.1. Paralelismo de grano grueso	44
2.4.1.2. Paralelismo de grano fino	44
2.4.2. Calendarización de flujos de trabajo	46
2.5. Tuberías de procesamiento de datos	46
2.6. Big data	48
2.6.1. Procesamiento por lotes	49
2.6.2. Procesamiento en tiempo real	50
2.6.3. Procesamiento híbrido	52
2.7. Red de distribución de contenidos	53
2.8. Infraestructuras para el manejo de servicios	54
2.8.1. Clústeres de alto rendimiento	54
2.8.2. Grid computing	55
2.8.3. Cómputo en la nube	55
2.8.4. Cómputo borde-niebla-nube	58
2.8.5. Mecanismos de despliegue de aplicaciones en la infraestructura	60
2.9. Contenedores virtuales	61
2.9.1. Docker	61
2.9.1.1. Docker Compose	63
2.9.1.2. Docker Swarm	65
2.9.2. Kubernetes	66
2.9.3. Singularity	67
2.10. Cómputo sin servidor	67
2.11. Resumen	70
3. Estado del arte	73
3.1. Manejo de contenedores virtuales en la literatura	74
3.2. Herramientas para el manejo de microservicios	75
3.3. Herramientas para diseñar estructuras de procesamiento	77
3.4. Manejo de requerimientos no funcionales en estructuras de procesamiento	79
3.5. Soluciones agnósticas para manejar el ciclo de vida de los productos digitales	83
3.6. Manejo de cuello de botellas en estructuras de procesamiento	87
3.7. Discusión	90
3.7.1. Manejo del ciclo de vida de las estructuras de procesamiento	90

3.7.2. Análisis de independencia de herramientas para el manejo de estructuras de procesamiento	95
4. Método de procesamiento continuo para la construcción y manejo de estructuras de procesamiento agnósticas de la infraestructura	97
4.1. Modelo de construcción de estructuras de procesamiento basadas en bloques autosimilares y autocontenidos	99
4.1.1. Manejo de aplicaciones como piezas de rompecabezas	99
4.1.2. Acoplamiento de piezas de software para crear estructuras de procesamiento	102
4.1.3. Modelando estructuras de procesamiento agnósticas de la infraestructura como un rompecabezas	103
4.1.4. Esquemas de preparación de datos para el manejo de requerimientos no funcionales en productos digitales	106
4.2. Modelo de comunicación y programación de infraestructuras como código	108
4.2.1. Flujos de datos intraorganizacionales	111
4.2.2. Flujos de datos interorganizacionales	112
4.2.3. Malla de servicios para el manejo de servicios	113
4.3. Modelo de gestión de tareas/datos y paralelismo implícito basado en monitoreo, manejo de colas y autoescalamiento	116
4.3.1. Esquema de paralelismo implícito	116
4.3.2. Algoritmo de balanceo de carga de datos	120
4.3.3. Identificación y manejo de cuellos de botella en un rompecabezas	123
4.3.3.1. Esquema de monitoreo continuo de servicios desplegados en múltiples infraestructuras	124
4.3.3.2. Identificación de cuellos de botella en un rompecabezas	126
4.3.3.3. Esquema de mitigación de cuellos de botella	127
4.4. Arquitectura y principios de diseño	130
4.4.1. Diseño y manejo de rompecabezas con la arquitectura	134
4.4.2. Clientes de acceso a la arquitectura	136
4.4.3. Implementación de un prototipo basado en la arquitectura del método propuesto	137
5. Evaluación experimental: metodología y resultados	139
5.1. Metodología de experimentación	139
5.2. Materiales, métricas y ambiente de pruebas	140
5.2.1. Métricas	140
5.2.2. Conjuntos de datos	142
5.2.3. Infraestructura	143
5.3. Estudio de caso 1: Procesamiento de registros climatológicos	144
5.3.1. Resultados experimentales	145
5.3.2. Experimento 1: Evaluación autocomparativa de soluciones paralelas	146
5.3.3. Experimento 2: Comparación de PuzzleMesh con herramientas del estado del arte	148
5.4. Estudio de caso 2: Manejo de imágenes satelitales	150

5.4.1. Experimento 1: Evaluación de balanceadores de carga incluidos en patrones paralelos	152
5.4.2. Experimento 2: comparación directa con herramientas del estado del arte	154
5.5. Estudio de caso 3: manejo y procesamiento de datos médicos	157
5.5.1. Diseño de experimentación y resultados experimentales	159
5.5.1.1. Experimento 1: Midiendo los costos de despliegue en diferentes plataformas	161
5.5.1.2. Experimento 2: Evaluación del rendimiento de los patrones paralelos incluidos en PuzzleMesh.	162
5.5.1.3. Experimento 3: Análisis estadístico	163
5.5.1.4. Experimento 4: Analizando los costos de preparación de los datos .	165
5.5.1.5. Experimento 5: Comparación con una solución ad-hoc.	166
5.5.1.6. Experimento 6: Comparación del rendimiento de PuzzleMesh con una herramienta de la literatura	167
5.5.1.7. Experimento 7: Manejo de cuellos de botella en tiempo de ejecución	168
5.5.1.8. Experimento 8: Reutilizando los esquemas de preparación de datos para transportar datos de ECG.	171
5.6. Análisis de la experiencia de usuario	172
6. Conclusiones, limitaciones y trabajo futuro	175
6.1. Conclusiones	175
6.2. Limitaciones	180
6.3. Trabajo futuro	181
A. Publicaciones	183
B. Ejemplo de un archivo de configuración de PuzzleMesh	251

Índice de Figuras

1.1.	Representación conceptual de las etapas para el manejo del ciclo de vida de los productos digitales.	3
1.2.	Ejemplo de una estructura de procesamiento para el manejo del CVPD.	6
1.3.	Ejemplos de patrones utilizados en flujos de trabajo.	7
1.4.	Ejemplo de un conjunto de bloques acoplados en un sistema de aplicaciones utilizando modelos de comunicación y procesamiento.	9
1.5.	Posibles dependencias que puede generar una estructura de procesamiento o sus componentes con la infraestructura, software o plataforma.	13
1.6.	Representación conceptual del proceso de materialización de una estructura de procesamiento abstracta, creada en tiempo de diseño, en un sistema de aplicaciones en tiempo de despliegue y ejecución.	22
1.7.	Clasificación de soluciones de acuerdo a su nivel de agnosticismo, control, autonomía, planeación y manejo operacional.	24
1.8.	Representación conceptual del proceso de convertir el diseño de una estructura de procesamiento agnóstica en un sistema agnóstico de aplicaciones.	27
1.9.	Diseño conceptual de la metodología seguida en esta tesis para crear estructuras de procesamiento agnósticas de la infraestructura.	29
1.10.	Ejemplo conceptual de una estructura autosimilar.	30
1.11.	Metodología de investigación propuesta.	32
2.1.	Arquitectura de un motor de flujos de trabajo propuesto por Rodriguez & Buyya [174].	43
2.2.	Representación conceptual de un flujo de trabajo compuesto por subflujos.	44
2.3.	Representación conceptual de un flujo de trabajo con paralelismo de datos.	44
2.4.	Representación conceptual de un flujo de trabajo con paralelismo de tareas.	45
2.5.	Tubería tradicional de procesamiento de datos. Imagen extraída de https://hazelcast.com/glossary/data-pipeline/	47
2.6.	Tubería de datos para el procesamiento de datos en tiempo real. Imagen extraída de https://hazelcast.com/glossary/data-pipeline/	47
2.7.	Comparación de los modelos ETL y ELTTT. Imagen extraída de [148].	49
2.8.	Comparación del procesamiento en lotes y en tiempo real. Imagen extraída de https://aws.amazon.com/blogs/bigdata/implement-serverless-log-analytics-using-amazon-kinesis-analytics/	50
2.9.	Representación conceptual de Amazon Kinesis. Imagen extraída de https://aws.amazon.com/blogs/big-data/implementserverless-log-analytics-using-amazon-kinesis-analytics/	51
2.10.	Arquitectura Lambda. Imagen extraída de Hausenblas, M., & Bijnens, N. (2015) [90].	52
2.11.	Ejemplo de una estructura de procesamiento desplegada en el borde-niebla-nube.	60
2.12.	Ejemplo de un archivo Dockerfile.	62

2.13. Ejemplo de un archivo YML utilizando en Docker Compose.	64
2.14. Clúster de Docker Swarm. Imagen extraída de https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/	65
2.15. Comparación del control del desarrollador sobre el código de las aplicaciones y la infraestructura en cómputo sin servidor, IaaS y SaaS. Imagen extraída de [17].	69
2.16. Arquitectura sin servidor. Imagen extraída de [89].	70
3.1. Arquitectura de Istio. Imagen extraída de https://istio.io/latest/about/service-mesh/	76
3.2. Comparación del manejo de requerimientos no funcionales en diferentes soluciones del estado del arte.	82
3.3. Clasificación de herramientas del estado del arte que implementan técnicas para mitigar cuellos de botella. <i>Definido por el usuario</i> : el usuario especifica los parámetros para mitigar los cuellos de botella. <i>Definidos por el sistema</i> : el sistema identifica y decide como mitigar los cuellos de botella.	87
3.4. Grado de independencia de diferentes herramientas del estado del arte.	95
4.1. Diseño de una pieza de rompecabezas en PuzzleMesh: <i>a</i>) Pila arquitectónica de sus componentes y <i>b</i>) su representación conceptual en la metáfora de rompecabezas.	100
4.2. Proceso de diseño y manejo de piezas de software en PuzzleMesh.	101
4.3. Representación conceptual de un rompecabezas construido en la forma de una tubería.	104
4.4. Representación conceptual de <i>a</i>) un rompecabezas creado mediante el acoplamiento de piezas, <i>b</i>) su representación como un DAG y <i>c</i>) y su notación en PuzzleMesh.	106
4.5. Ejemplo de un esquema de recuperación y preparación de datos incluidos en las interfaces de entrada y salida de una pieza.	107
4.6. Representación conceptual de un meta-rompecabezas compuesto de dos rompecabezas.	110
4.7. Ejemplo de un flujo de datos intraorganizacional en un hospital.	110
4.8. Ejemplo de un flujo de datos interorganizacional construido con PuzzleMesh para el intercambio de datos entre hospitales.	113
4.9. Representación conceptual del repositorio de piezas disponibles en la malla de servicios.	114
4.10. DAG de un rompecabezas que incluye un patrón paralelo.	117
4.11. Representación conceptual de una pieza de software manejada como un patrón de paralelismo.	118
4.12. Métricas observadas y recolectadas en un rompecabezas.	124
4.13. Esquema de monitoreo del rendimiento de piezas en un rompecabezas.	126
4.14. Manejo de colas utilizando patrones de paralelismo.	128
4.15. Esquema de autoescalamiento de los patrones de paralelismo incluidos en las piezas.	129
4.16. Diagrama de flujo del proceso de autoescalamiento en un rompecabezas.	131
4.17. Arquitectura para el manejo de piezas, rompecabezas y meta-rompecabezas.	132
4.18. Pila de servicios del cliente implementado para interactuar con la arquitectura.	137
5.1. Estructura de procesamiento para el manejo de registros meteorológicos.	144
5.2. Tiempo de respuesta observado durante el procesamiento de datos meteorológicos con diferentes configuraciones creadas con PuzzleMesh.	146

5.3. Configuración general de cada herramienta utilizada para conducir el segundo experimento de este estudio de caso.	148
5.4. Tiempo de respuesta producido por las diferentes herramientas evaluadas. MH: Número máximo de hilos configurado. NC: Número de núcleos configurado en HTCondor. NM: Número máximo de núcleos configurado en Makeflow.	150
5.5. Representación conceptual de la estructura de procesamiento desarrollada para conducir este estudio de caso.	151
5.6. Porcentaje de error en el balanceo de carga observado con tres diferentes平衡adores de carga.	153
5.7. Tiempo de respuesta observado durante el procesamiento de imágenes satelitales en a) un clúster local y b) la nube.	156
5.8. Meta-rompecabezas diseñado para el manejo y procesamiento de datos médicos. . .	157
5.9. Tiempo de respuesta observado durante el despliegue del Rompecabezas 1 en tres diferentes infraestructuras.	160
5.10. Tiempo de respuesta y desempeño observado para el procesamiento de 25 estudios médicos con el Rompecabezas 1	161
5.11. Tiempo de respuesta y rendimiento observados durante el procesamiento de 1104 imágenes médicas con un rompecabezas para la identificación de tumores en pulmón. .	163
5.12. Tiempo de respuesta observado al cargar los datos desde el nodo de procesamiento hasta el nodo de almacenamiento.	165
5.13. Porcentaje de incremento y reducción en el tiempo de respuesta de PuzzleMesh y una solución ad-hoc.	166
5.14. Comparación directa de PuzzleMesh con Makeflow.	168
5.15. Rendimiento observando durante el procesamiento de imágenes médicas con 3 piezas.	169
5.16. Rendimiento observando durante el procesamiento de imágenes médicas con 4 piezas.	170
5.17. Tiempo observado durante la codificación cifrado y carga de 1000 trazas de ECG. .	171
5.18. Análisis comparativo de las acciones requeridas para configurar, desarrollar y ejecutar una solución utilizando diferentes herramientas para construir sistemas de aplicaciones.	173

Índice de Tablas

3.1. Manejo de contenedores virtuales en diferentes herramientas disponibles en la literatura	74
3.2. Comparación cualitativa de diferentes herramientas del estado del arte para construir estructuras de procesamiento. ‡ representa que la herramienta utiliza Condor o MPI para agregar paralelismo a las estructuras de procesamiento.	79
3.3. Comparación cualitativa de soluciones agnósticas para manejar el ciclo de vida de los productos digitales.	84
3.4. Herramientas identificadas en la literatura para construir y manejar el ciclo de vida de las estructuras de procesamiento.	91
5.1. Principales características de los tres conjuntos de productos digitales utilizados para conducir cada estudio de caso.	142
5.2. Características de los equipos, máquinas virtuales y contenedores utilizados para experimentación.	143
5.3. Resumen de los experimentos diseñados para cumplimentar los requerimientos no funcionales de portabilidad, manejo de la heterogeneidad, reusabilidad, eficiencia y confiabilidad.	160
5.4. Resultados de la comparación a pares de cada configuración paralela evaluada.	164

Índice de Algoritmos

1.	Algoritmo de ejecución de un patrón en una pieza.	119
2.	Algoritmo de balanceo de carga fuera de línea.	121
3.	Algoritmo de balanceo de carga en línea.	122
4.	Algoritmo para la identificación de un cuello de botella.	127

Resumen

Método de procesamiento continuo para la construcción y manejo de estructuras de procesamiento agnósticas de la infraestructura

por

Dante Domizzi Sánchez Gallegos

Cinvestav Unidad Tamaulipas

Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2023

Dr. José Luis González Compeán, Director

Los motores de flujos de trabajo (p. ej. Parsl, Makeflow y Pegasus) y los gestores de tuberías para el procesamiento de datos (p. ej. Nextflow y Jenkins) son marcos de *diseño y construcción* usados habitualmente para crear *estructuras de procesamiento* (p. ej. tuberías, flujos de trabajo o patrones de diseño), las cuales definen las reglas para integrar múltiples aplicaciones como una sola pieza de software. Tradicionalmente, estos marcos de construcción han sido utilizados para diseñar software de gran escala (p. ej. tuberías de entrega continua e integración continua, así como tuberías de *big data*) y, recientemente, para diseñar sistemas complejos de procesamiento de datos (p. ej. sistemas para procesos de toma de decisiones y sistemas de ciencia de datos). Estos sistemas no solo coordinan la ejecución de las aplicaciones que los conforman, sino que también orquestan el manejo, almacenamiento e indexamiento de las múltiples versiones de datos producidas por sus aplicaciones (llamados en la literatura *productos/activos digitales*). Estos productos pueden, potencialmente, ser visualizados, consumidos y/o compartidos por múltiples usuarios finales, creando de esta forma un ciclo de vida. Estas tareas de gestión suelen ser delegadas, en *tiempo de ejecución*, a gestores de recursos computacionales y de carga de trabajo (p. ej. HTCondor o Slurm). Tanto los sistemas de aplicaciones como los gestores de recursos son comúnmente *desplegados* sobre el mismo contexto computacional. Por ejemplo, sobre servidores de centros de datos o máquinas virtuales/físicas en sitios de altas prestaciones (p. ej. Grid y clústeres de servidores). Sin embargo,

en años recientes, estas estructuras se han comenzado a desplegar en la nube, lo cual ha llevado a que las organizaciones deleguen el control del manejo de procesos, y sobre todo de datos de estos sistemas de aplicaciones, a los proveedores de servicios en la nube. Esto produce una dependencia funcional con la infraestructura, plataforma y software de dichos proveedores. Adicionalmente, la interconexión entre los gestores de recursos y los sistemas de aplicaciones termina produciendo, en tiempo de ejecución, cuellos de botella durante el manejo de las etapas asociadas al ciclo de vida de productos digitales. En este trabajo de tesis, se propone agregar la propiedad de agnosticismo a las estructuras de procesamiento como una solución para mitigar los efectos relacionados a la dependencia de estas con las plataformas e infraestructuras donde se despliegan, así como la potencial ruptura de coordinación del manejo de los datos intercambiados por aplicaciones/procesos en tiempo de ejecución. Para agregar esta propiedad a las estructuras de procesamiento, se propone un modelo de encapsulación de bloques de construcción autocontenidos (los bloques incluyen un ambiente de trabajo que garantiza la ejecución de las aplicaciones para eliminar dependencias con plataformas de software) y autosimilares (esto posibilita la interconexión de aplicaciones en una estructura para evitar la dependencia con lenguajes de programación). Además, un modelo de comunicación integrado a un modelo programación orientado al código permite resolver, en tiempo de despliegue, los conflictos del acoplamiento de estos bloques con bloques de orquestación de datos, tales como almacenamiento y logística de entrega/recepción de datos, lo cual reduce la dependencia con la infraestructura. Por otro lado, un modelo de procesamiento de paralelismo implícito evita delegar tareas a gestores de recursos y reduce, en tiempo de ejecución, los efectos de cuellos de botella producidos por las aplicaciones dentro de una estructura de procesamiento, los cuales pueden reaccionar a cambios en la carga de trabajo o en la infraestructura. Un prototipo basado en el método propuesto fue evaluado en diferentes estudios de caso basados en aplicaciones para el procesamiento de imágenes satelitales, contenidos médicos (señales de electrocardiograma y tomografías), y datos meteorológicos. En la evaluación, se demostró la flexibilidad del método propuesto para construir diferentes estructuras de procesamiento agnósticas para el manejo de ciclos de vida de productos digitales. Además, la evaluación reveló la eficiencia del método propuesto comparado con motores de flujos de trabajo y gestores de tuberías disponibles en el estado del arte.

Abstract

A continuity processing method for building and managing infrastructure-agnostic processing structures

by

Dante Domizzi Sánchez Gallegos

Cinvestav Unidad Tamaulipas

Research Center for Advanced Study from the National Polytechnic Institute, 2023

Dr. José Luis González Compeán, Advisor

Workflow engines (e.g., Parsl, Makeflow, and Pegasus) and data processing pipeline managers (e.g., Nextflow and Jenkins) are design and construction frameworks typically used to create *processing structures* (e.g., pipelines, workflows, or design patterns) that integrate multiple applications in a single software system. Traditionally, these frameworks are used to design large-scale software (e.g., continuous integration and continuous delivery pipelines as well as big data pipelines) and recently to design complex data processing systems (e.g., decision-making systems and data science systems). These systems not only coordinate the execution of the multiple applications that are part of them, but also orchestrate the management, storage, and indexation of the multiple data versions produced by these applications (called in the literature *digital products/assets*). These products can be potentially visualized, consumed, and/or shared by multiple end-users, creating a lifecycle. The management tasks are usually delegated, during *execution time*, to computational resources and workload managers (e.g., HtCondor or Slurm). Both applications and resources managers are usually *deployed* over the same computational context. For example, over servers in data centers or virtual/physical machines in high-performance sites (e.g., Grid and server clusters). Nevertheless, in recent years, these structures have been started to be deployed in the cloud, which results in organizations delegating the management control of processes and mainly of data produced by these application systems, to the service providers in the cloud. This produces a functional dependence on the infrastructure, platform, and software of these providers. Additionally, the interconnection

between resource managers and application systems produces, during execution time, bottlenecks in the management of the stages associated with the digital product lifecycle. In this thesis is proposed to add the agnosticism property to the processing structures as a solution to mitigate the effects related to their dependency on the platforms and infrastructures where they are deployed, as well as the potential interruption in the coordination of the management of data exchanged by the applications and processes during execution time. To add this property of agnosticism to a processing structure, it is proposed a model where applications are encapsulated into self-contained (to remove the dependencies with the programming language and the platform) and self-similar building blocks (to remove the dependency on the infrastructure). A communication model integrated with a code-oriented programming model resolves, during deployment time, the issues related to the coupling of these blocks with data orchestration blocks (storage and logistic of data delivery/retrieval). An implicit parallelism processing model avoids delegating tasks to resource managers and reduces, during execution time, the effects of bottlenecks produced by the applications considered in a processing structure, which can react to changes in the workload or the infrastructure. A prototype based on the proposed method was evaluated in different case studies based on application systems for the processing of satellite imagery, medical contents (electrocardiogram signals and tomographies), and meteorological data. During the evaluation was revealed the flexibility of the proposed model to build different agnostic processing structures to manage different digital product lifecycles. Moreover, the evaluation reveals the efficiency of the proposed method compared with workflow engines and pipeline managers available in the related work.

Nomenclatura

API	Interfaz de programación de aplicaciones.
IaC	Infraestructura como código.
CVPD	Ciclo de vida de los productos digitales.
CVEP	Ciclo de vida de las estructuras de procesamiento.
CI/CD	Integración continua/entrega continua.
CV	Contenedor virtual.
IaaS	Infraestructura como servicio.
PaaS	Plataforma como servicio.
SaaS	Software como servicio.

1

Introducción

El volumen de datos producidos por las organizaciones y la industria ha crecido exponencialmente en los años recientes¹. Estudios estiman [170] que los datos producidos por aplicaciones en el dominio de la medicina crecerán a un ritmo de un 36% entre los años 2018 y 2025. Este comportamiento también se ha observado en otras industrias, tales como la manufactura, los servicios financieros y los medios de entretenimiento [170].

Actualmente, las organizaciones han comenzado a procesar estos cúmulos de datos con el fin de extraer información útil con la cual crear activos de información llamados *productos digitales*, los cuales puedan ser utilizados para soportar sus procesos de toma de decisiones [53, 71]. Los productos digitales, creados en un formato electrónico (p. ej. XLSX, PDF, HTML, DOCX, etc.), pueden ser distribuidos para su consumo por múltiples usuarios. En el presente trabajo de tesis, se estudian productos digitales, incluyendo contenido utilizado principalmente como soporte de procesos de toma

¹Se estima que el volumen de datos a nivel mundial crecerá desde 33 zettabytes en 2018 hasta 175 zettabytes en 2025 [170].

de decisiones, tales como diagnósticos, pronósticos o intervenciones [71]. Ejemplos de este tipo de producto son los contenidos médicos con anotaciones, imágenes satélites incluyendo índices sobre objetos territoriales, interpretación de señales producidas por sensores, mapas que incluyen patrones, gráficas indicando tendencias, reportes de revisión de contratos o contenidos multimedia (audio o video) [199, 213].

1.1 Ciclo de vida de los productos digitales

Los productos digitales, tales como los productos de bienes y servicios tradicionales [97], son creados, manufacturados, intercambiados, entregados y/o consumidos durante un ciclo de vida. En la literatura se pueden encontrar dos tipos de aplicación del concepto: *ciclo de vida de los productos digitales* (CVPD). La primera aplicación de este concepto apareció en el dominio comercial, donde un CVPD hace referencia a la evolución de un producto desde que es diseñado y desarrollado hasta que es lanzado al público, y finalmente es utilizado o vendido [225]. Ejemplos de este tipo de producto son aquellos relacionados principalmente con el entretenimiento, vigilancia y desarrollo de software de gran escala, los cuales siguen el modelo de integración continua/entrega continua (CI/CD, por sus siglas en inglés) [191, 225]. La segunda aplicación del concepto CVPD se puede encontrar en el área de ciencia de datos [71], donde un ciclo comienza a partir de la adquisición de datos en su formato crudo (p. ej. utilizando sensores, dispositivos médicos o bases de datos) [51], pasando por su transformación (mediante el uso de aplicaciones de analítica, minería de datos o inteligencia artificial) [165], y terminando en su uso/consumo por los usuarios finales (generalmente por sistemas de visualización/vigilancia usados en procesos de toma de decisiones) [175].

El procesamiento de análisis de imágenes médicas para detectar tumores es un ejemplo clásico de un CVPD [15, 81] en sistemas de ciencia de datos. En este tipo de escenario, un dispositivo médico captura datos en forma de imágenes (etapa de adquisición), después las imágenes son

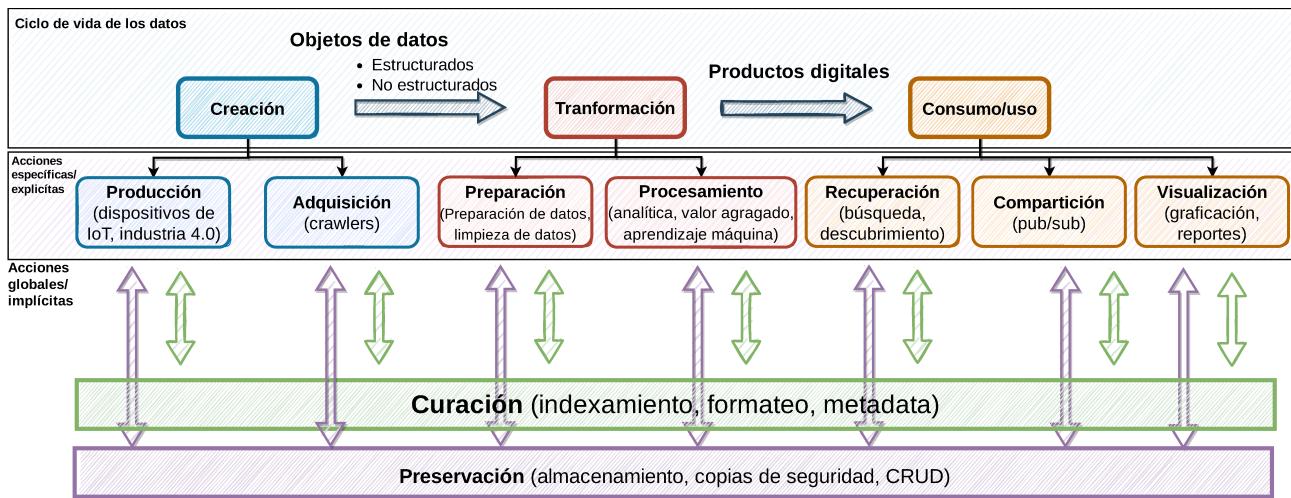


Figura 1.1: Representación conceptual de las etapas para el manejo del ciclo de vida de los productos digitales.

colocadas en un formato manejable para un algoritmo dado (etapa de preparación), y finalmente los contenidos son injectados a una aplicación de aprendizaje máquina (etapa de procesamiento) la cual devuelve una probabilidad de que el contenido analizado se corresponda con algún tipo de cáncer. Esta probabilidad es observada por el operario del sistema para tomar una decisión (etapas de compartición y visualización). Como se puede ver, esto crea un flujo de datos que va desde los dispositivos médicos hasta el experto, pasando por algoritmos de aprendizaje máquina [165, 185]. De esta manera, a través de cada etapa, se crea un conjunto de diferentes versiones del producto crudo (imágenes en formato DICOM o *Digital Imaging and Communication In Medicine*, por sus siglas en inglés [150]), imágenes en formato PNG creadas en el proceso de preparación, imágenes marcadas en el proceso de aprendizaje/modelado e imágenes incluyendo un lugar de ubicación de una masa con una probabilidad de que aquello identificado sea un tumor del tipo aprendido.

En este trabajo de tesis, se estudia la segunda aplicación del concepto CVPD, en la cual se suelen incluir etapas de creación, transformación y consumo/uso. Para formalizar este concepto, se incluyeron etapas adicionales para el manejo de datos (p. ej. los productos generados por etapas intermedias), las cuales son mostradas en la Figura 1.1 como *etapas explícitas e implícitas*.

Las *etapas explícitas* son todas aquellas requeridas y declaradas por las organizaciones para gestionar la creación de sus productos digitales. Se espera que cada una de estas etapas produzca una nueva versión del producto digital original [71]. Esto significa que, en la práctica, las organizaciones pueden elegir las aplicaciones que son de interés para producir un producto final con ciertas características a través de un CVPD. Por ejemplo, en la etapa de creación se pueden activar/elegir fases para la producción y/o adquisición de productos digitales en las cuales se utilizan aplicaciones para recolectar datos (p. ej. sensores o dispositivos desplegados en ambientes reales o bases de datos de registros históricos de eventos), los cuales son depositados en una fuente de datos (p. ej. una piscina o almacén de datos) [51, 182]. En la etapa de transformación se contemplan fases para el preprocesamiento y procesamiento de los productos digitales, utilizando aplicaciones de preparación y limpieza de datos en el caso de preprocesamiento, y aplicaciones de análisis de datos y aprendizaje máquina en el caso del procesamiento de los datos [138, 185, 199]. En la etapa de consumo/uso se contemplan fases para la recuperación (básicamente implementada mediante motores de búsqueda) y compartición (implementadas mediante herramientas de publicación/suscripción) de los productos digitales. Estas fases o subetapas suelen considerar aplicaciones para la visualización, búsqueda y descubrimiento de los productos digitales por parte de los usuarios finales [73, 175]. Se espera también que cada etapa produzca una versión diferente del producto, la cual podría ser necesaria para otra etapa y que, a su vez, lo use como insumo de entrada con el fin de producir otra versión del producto entrante.

Las *etapas implícitas* incluyen fases para la curación y preservación de las diferentes versiones de los productos digitales y sus metadatos. La curación de datos es el proceso por el cual diversas fuentes de datos son recolectadas en repositorios, permitiendo el manejo de los metadatos de los productos digitales, incluyendo su indexación y su transformación a un formato coherente [72]. La preservación de datos incluye aquellas tareas que permiten conservar los productos digitales y todas sus subversiones a través del tiempo mediante el uso de sistemas de almacenamiento (p. ej. en la nube) [188]. Además, en la etapa de preservación se incluyen tareas asociadas a requerimientos no funcionales, tales como seguridad [169, 196], integridad [227], o respaldos de los productos digitales

para tolerar fallas en los nodos de almacenamiento [29, 95]. De la misma forma, se requiere crear, actualizar, eliminar y remover productos digitales de los sistemas de almacenamiento [205].

1.2 Antecedentes y motivación

Como se ha descrito previamente, cuando las organizaciones pretenden implementar un sistema de procesamiento de grandes volúmenes de datos para gestionar un CVPD para servir de soporte a procesos de toma de decisiones, las organizaciones terminan implementando o utilizando tantas aplicaciones como tareas requeridas en cada etapa del CVPD. Esto significa que las organizaciones terminan lidiando con un ecosistema heterogéneo de herramientas, la cuales deberían ser consolidadas como un solo sistema [55, 118]. Este ecosistema heterogéneo considera aplicaciones que podrían haber sido desarrolladas en diferentes lenguajes de programación, dependiendo de las necesidades de la organización [8]. Por ejemplo, una organización podría utilizar una aplicación implementada en Java para anonimizar imágenes en formato DICOM, y posteriormente depositarlas en un sistema PACS (del inglés, *Picture Archiving and Communication System*) [129] para su almacenamiento, o utilizar una herramienta desarrollada en Python con Tensorflow [156] para identificar tumores en las imágenes médicas [157].

En este contexto, diseñar sistemas de aplicaciones que manejen un CVPD es un proceso que involucra muchos retos. Lo anterior se debe a que es necesario establecer el acoplamiento de las aplicaciones elegidas por los diseñadores para garantizar que los productos son manejados por las etapas requeridas para conseguir las cualidades o los tipos de productos que buscan las organizaciones.

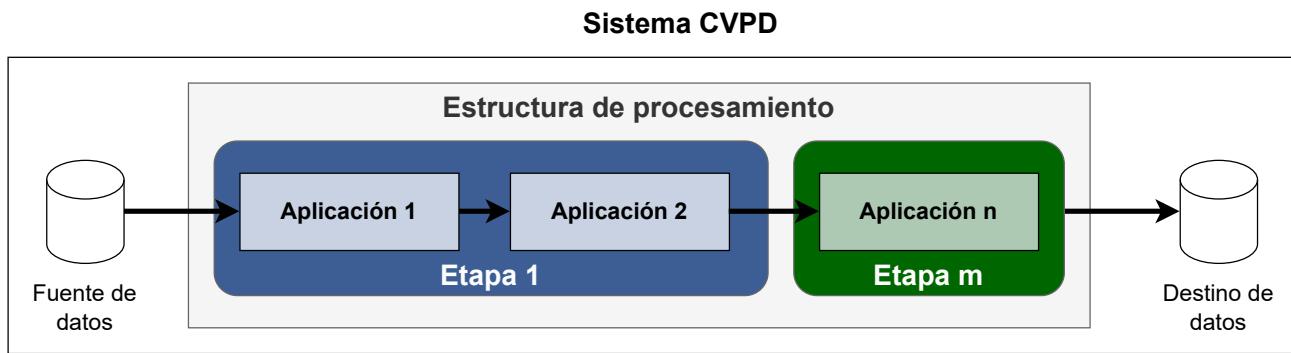


Figura 1.2: Ejemplo de una estructura de procesamiento para el manejo del CVPD.

1.2.1 Estructuras de procesamiento para el manejo de CVPD

En la literatura, los problemas de acoplamiento antes mencionados se han resuelto mediante el modelado de los CVPD como estructuras de procesamiento. Las estructuras de procesamiento se definen mediante un conjunto de reglas que serán utilizadas, en tiempo de diseño, para garantizar la integración de múltiples aplicaciones requeridas por las etapas de un CVPD con el fin de convertirlas en una sola pieza de software, la cual es presentada al usuario final como un sistema de aplicaciones o un servicio [21, 206] (ver Figura 1.2). Por lo tanto, una estructura de procesamiento es definida como un conjunto de reglas y aplicaciones utilizadas para manejar y procesar un conjunto de productos digitales durante su ciclo de vida. La idea básica es modelar las aplicaciones de las etapas de procesamiento como nodos/filtros, mientras que el intercambio de las diferentes versiones de los productos digitales es modelado como aristas/conectores. Es por esto que las estructuras clásicas para la integración de aplicaciones para procesar datos en un CVPD son los flujos de trabajo (workflows) [21, 174] [120], las tuberías (pipelines) [142], los grafos acíclicos dirigidos (DAG, por sus siglas en inglés) [110, 201] y los *patrones* de diseño [155, 173].

Los flujos de trabajo son estructuras de procesamiento cuyo objetivo es coordinar la ejecución de un conjunto de aplicaciones organizadas en forma de etapas [21, 120]. Este modelo de construcción se ajusta al procesamiento requerido para el manejo de diferentes CVPD, ya que las etapas de un

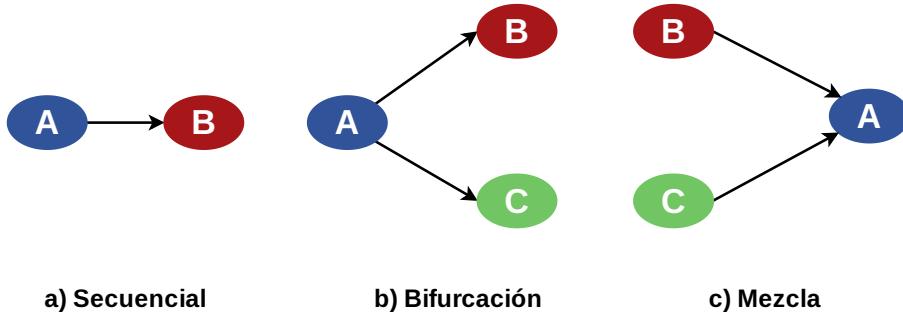


Figura 1.3: Ejemplos de patrones utilizados en flujos de trabajo.

fluo de trabajo se pueden organizar siguiendo diferentes patrones [21]. En la Figura 1.3, se muestran tres ejemplos de patrones de diseño que pueden ser creados en un flujo de trabajo, los cuales son descritos a continuación:

Secuencial: un conjunto de etapas del flujo de trabajo son ordenadas de forma secuencial ($A \rightarrow B \rightarrow C$), en donde cada una de las etapas es ejecutada al completar la etapa anterior. Por ejemplo, la etapa B solo será ejecutada hasta que la etapa A sea completada.

Bifurcación: Los datos producidos por una etapa son entregados a múltiples etapas. Cada una de estas etapas implementa diferentes aplicaciones para procesar los datos y producir diferentes versiones de estos.

Mezcla: La salida de distintas etapas convergen en una sola, la cual recibe los datos de las etapas productoras para procesarlos.

Los motores de flujos de trabajo (p. ej. Parsl [16], Makeflow [5], Pegasus [57], Triana [202] o DagOnStar [139]) se popularizaron entre las organizaciones como soluciones para modelar CVPDs complejos como flujos de trabajo convertidos en piezas de software, las cuales incluyen múltiples etapas y múltiples aplicaciones por cada etapa. Por ejemplo, estos motores se usan para crear flujos de trabajo que regulen el procesamiento ejecutado por sistemas que son utilizados en ambientes de cómputo de altas prestaciones (HPC, por sus siglas en inglés) para procesar grandes conjuntos de datos. Este tipo de sistemas generalmente tienen el objetivo de conducir experimentos que requieren

el despliegue y ejecución de múltiples procesos [8, 131].

Las tuberías de procesamiento de datos [60, 192], en cambio, son principalmente usadas por las organizaciones para integrar software y ponerlo, automáticamente, en producción mediante un concepto llamado *Entrega Continua/Integración Continua* (CD/CI, por sus siglas en inglés) [191, 225]. Estos marcos de trabajo crean estructuras de procesamiento que serializan múltiples etapas, las cuales son organizadas siguiendo una relación *productor-consumidor*. En esta relación, una etapa produce datos (productor) y los entrega a la siguiente (consumidor) para su posterior procesamiento [128]. Cada etapa en la tubería puede procesar múltiples productos digitales de forma paralela, lo anterior se debe a que una etapa no debe esperar a que la etapa anterior procese todos los productos digitales de entrada para procesar sus propios datos de entrada. Este tipo de marcos de trabajo son principalmente utilizados para crear tuberías de extracción, transformación y carga (ETL, por su definición en inglés). Estas tuberías de procesamiento de datos son típicamente empleadas en escenarios de big data [147], y se han convertido en la solución de facto para procesar grandes volúmenes de datos y crear sistemas de ciencia de datos [71, 121].

Los gestores de tuberías para el procesamiento de datos, tales como Nextflow [60] y Jenkins [192] [46, 142, 225] son ejemplos de marcos de *diseño y construcción* utilizados en la literatura para crear *estructuras de procesamiento* basadas en el modelo de tuberías.

1.2.2 Modelos de despliegue de patrones de diseño para estructuras de procesamiento de datos

Los patrones de diseño son manejados como grafos acíclicos dirigidos y creados mediante la combinación de tuberías, bifurcaciones y flujos de trabajo [75, 127, 155, 173]. Esto significa que los componentes de una estructura de procesamiento (tubería o flujo de trabajo) pueden ser desplegados y manejados de forma paralela y distribuida siguiendo diferentes patrones de diseño (p. ej. patrón en

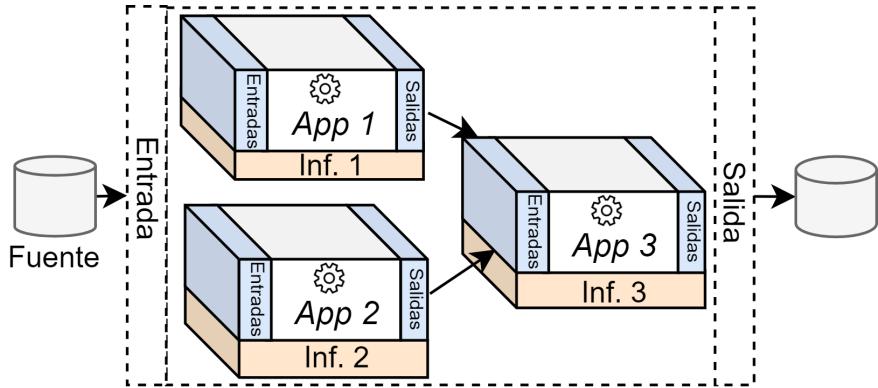


Figura 1.4: Ejemplo de un conjunto de bloques acoplados en un sistema de aplicaciones utilizando modelos de comunicación y procesamiento.

capas, cliente-servidor o maestro-esclavo) y paralelismo (p. ej. gestor/trabajador, divide&vencerás o fork-join) [155]. Estos patrones son diseñados, desplegados y ejecutados utilizando modelos de programación, comunicación y procesamiento.

1.2.2.1. Modelos de programación

En tiempo de diseño, los modelos de programación son utilizados por los diseñadores y desarrolladores para crear las reglas de comportamiento del patrón que usarán las estructuras de procesamiento. Esto se consigue mediante la declaración de las etapas, las fuentes y los destinos de los datos, así como de las configuraciones de cada aplicativo que será usado por cada etapa de un CVPD, lo cual dará como resultado la codificación de diferentes patrones de diseño y/o paralelismo [155, 173].

En la mayoría de los modelos de programación disponibles en la literatura para la creación de estructuras de procesamiento, las etapas son declaradas como funciones que reciben múltiples entradas y que procesan los datos llamando la ejecución de una aplicación, o ejecutando un segmento de código para producir resultados que son entregados a través de una salida o un conjunto de salidas. Por ejemplo, el motor de flujos de trabajo Parsl [16] incluye un modelo de programación basado en Python, el cual permite la creación de flujos de trabajo paralelos, declarando cada etapa

como una función que ejecuta código Python, o haciendo llamadas a aplicaciones externas para obtener los datos entrantes y entregar los resultados. Cada función contiene la declaración de estas operaciones de entrada y salida, así como de sus parámetros de paralelismo (número de hilos o núcleos a utilizar). Como resultado, los diseñadores de la estructura pueden permitir a los usuarios cambiar las aplicaciones de las etapas, así como definir los parámetros de dichas aplicaciones en tiempo de despliegue. En este tipo de modelo, los desarrolladores también declaran, mediante código, el orden de ejecución de las etapas que forman parte de la estructura de procesamiento. Esto significa que la programación del flujo garantiza una secuencia de ejecución de las aplicaciones, y además que, en tiempo de despliegue, los usuarios finales puedan ajustar o sintonizar los parámetros para obtener un rendimiento dado. De esta forma, el código de los modelos de programación para el diseño de patrones es interpretado y las etapas de la estructura de procesamiento, incluyendo las aplicaciones contempladas en estas etapas, son desplegadas en la infraestructura definida en el código.

1.2.2.2. Modelos de comunicación

Las aristas de las estructuras de procesamiento asociadas a la entrega/recepción de datos o productos digitales se despliegan mediante el uso de modelos de comunicación. Este tipo de modelo define las reglas de acoplamiento de las etapas siguiendo el DAG del patrón o tubería definidos en el modelo de programación. El concepto cliente-servidor se utiliza para modelar las llamadas (cliente) y respuestas (servidor) realizadas por las etapas que han sido acopladas en tiempo de diseño, lo cual permite a esas etapas intercambiar tantos datos o productos digitales como mensajes definidos por el modelo de programación. En práctica, un modelo de comunicación típicamente se implementa utilizando TCP/UDP sockets [112], REST APIs [27] o colas de mensajes [78, 104]. El resultado de implementar un modelo de comunicación es la interconexión de las etapas de una estructura de procesamiento mediante un canal virtual por el cual se entregarán los mensajes, los cuales se encuentran frecuentemente asociados a metadatos para describir las tareas que una etapa debe

realizar y las rutas desde las cuales obtendrá los datos entrantes (versión previa de un producto digital), así como donde se entregarán los datos resultantes (nueva versión del producto digital).

1.2.2.3. *Modelos de coordinación de procesamiento*

En tiempo de ejecución, los modelos de coordinación de procesamiento definen la inyección y manejo de carga de trabajo que será suministrada a las aplicaciones de cada etapa del CVPD, lo cual crea un flujo de datos a través de la estructura de procesamiento. Estos modelos establecen las reglas de procesamiento y entrega de datos entre etapas. Un ejemplo de este tipo de coordinación es el modelo ETL utilizado tradicionalmente en big data [147]. En este modelo, las tareas de coordinación del procesamiento de datos son modeladas como tres tipos de procesos: *i*) procesos de extracción (sistemas de lectura), *ii*) procesos de transformación (aplicaciones que se ejecutan en una etapa), y *iii*) procesos de carga (aplicaciones de entrega y/o escritura de datos o productos digitales).

En la Figura 1.4 se puede observar una representación conceptual del acoplamiento de tres aplicaciones utilizando modelos de comunicación y procesamiento. Como se puede observar, estos modelos permiten el intercambio de datos y mensajes entre diferentes aplicaciones distribuidas en diferentes infraestructuras. Para ello, se utiliza un conjunto de interfaces de entrada y salida que implementan un modelo de comunicación, el cual permite la generación de una única estructura coherente. Mientras que, el modelo de procesamiento se encarga de inyectar la carga de trabajo a través de cada aplicación considerada en la estructura.

1.2.3 Independencia y autonomía de las estructuras de procesamiento

La autonomía e independencia de los sistemas de aplicaciones creados mediante estructuras de procesamiento son propiedades conseguidas en función al grado de control que dichas estructuras

definieron, en tiempo de diseño, y que son aplicadas en tiempo de despliegue. El control de un sistema de aplicaciones se puede valorar en función de la producción de información que permita a las organizaciones conocer, en todo momento, las respuestas a preguntas tales como quién, cuándo, dónde y cómo, se utilizan los datos durante todo el CVPD [210]. A esto se le llama un *contexto computacional conocido* en el cual se espera que no se presenten errores tales como direccionamiento de rutas de almacenamiento y no disponibilidad de recursos asociados a rutas para la ejecución de aplicaciones, ya que estas permanecen estáticas y son conocidas y accesibles por las aplicaciones durante todo el tiempo de ejecución.

El control sobre la ejecución de las aplicaciones, la inyección de la carga de trabajo a dichas aplicaciones y la gestión del CVPD provee a los sistemas de aplicaciones con suficiencia que se traduce en *independencia* en tiempo de ejecución. La autonomía, en cambio, se consigue cuando las estructuras de procesamiento consideraron reglas para que los sistemas de aplicaciones posean la independencia (a través del control de procesamiento) de continuar realizando tareas de procesamiento sin requerir la intervención de componentes externos.

1.2.4 Dependencia de las estructuras de procesamiento con software, plataforma e infraestructuras

Es importante notar que tanto la autonomía como la independencia de los sistemas de aplicaciones son propiedades de un estado ideal, el cual está condicionado tanto por el diseño de las estructuras de procesamiento como por el contexto computacional en el que estas se despliegan. Esto significa que eventualmente los sistemas de aplicaciones presentarán grados de dependencias de diseño asociadas al software, plataforma e infraestructura.

En la Figura 1.5 se muestra la composición de estructuras de procesamiento para soportar el manejo de CVPD, así como las dependencias que estas estructuras pueden generar con la

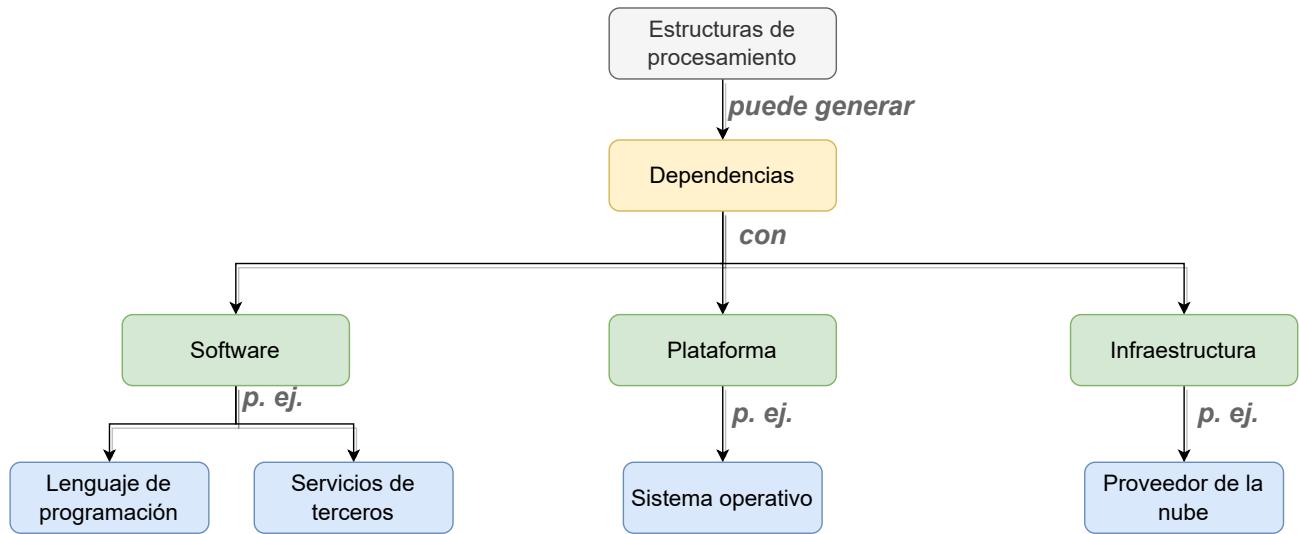


Figura 1.5: Posibles dependencias que puede generar una estructura de procesamiento o sus componentes con la infraestructura, software o plataforma.

infraestructura [171], la plataforma [94] o el software (p. ej. servicios de terceros y lenguajes de programación) [153].

La dependencia con el software se genera cuando la herramienta utilizada para diseñar y desplegar una estructura de procesamiento impone el uso de un lenguaje específico (p. ej. Python o C) para la programación de cada una de las aplicaciones que conforman a un sistema de aplicaciones para el procesamiento de CVPD. Esto está determinado por el *modelo de programación* de los motores de flujos de trabajo (p. ej. Makeflow, Parsl y DagOnStar permiten el manejo de aplicaciones diseñadas en múltiples lenguajes de programación, mientras que Metaflow [135] impone el uso de Python o R como lenguajes de programación) y los gestores de tuberías. Este tipo de restricción obstaculiza la libre interconexión de aplicaciones desarrolladas mediante el uso de diferentes lenguajes de programación. Por ejemplo, Gearbox [181] permite solo la interconexión de aplicaciones desarrolladas en el lenguaje de programación C, debido a que utiliza un esquema de intercambio de datos en memoria compartida que está limitado a aplicaciones C. Raftlib [24] es una herramienta para diseñar estructuras de procesamiento paralelo en tiempo real que impone el uso de C/C++ como lenguaje de programación para diseñar las etapas de las estructuras de procesamiento así como su interconexión.

La dependencia con la plataforma se refiere a las imposiciones de algunos motores y gestores para utilizar un sistema operativo o herramienta en específico para diseñar estructuras de procesamiento, así como desplegar y ejecutar los sistemas de aplicaciones. Pero especialmente se refiere a la imposición de los motores y gestores de una plataforma para la gestión de recursos computacionales para soportar el despliegue y ejecución de sus estructuras de procesamiento (p. ej. Pegasus y Makeflow requieren la instalación de HTCondor, Brigade [32] requiere la instalación de Kubernetes y Amazon Kinesis depende de otros servicios de Amazon [211]). Estas plataformas definen y reservan los recursos que serán utilizados por los procesos de los sistemas de aplicaciones. El balanceo y/o despacho de carga y la gestión de solicitudes ante los sistemas operativos de las infraestructuras son algunas de las tareas que son delegadas por los sistemas de aplicaciones a este tipo de herramientas. Este requerimiento resulta crítico para motores y gestores, ya que los sistemas de aplicaciones creados mediante sus estructuras de procesamiento no solo deben coordinar la ejecución de los procesos, sino que también deben orquestar el manejo, almacenamiento e indexamiento de las múltiples versiones de datos producidas por sus aplicaciones en un CVPD, lo cual no se debería traspasar a los programadores, ya que esto representaría un desafío para ellos. Este tipo de restricción impone una dependencia de los sistemas de aplicaciones con el prerequisito de tener instalada una plataforma de gestión para garantizar el funcionamiento de sus aplicaciones durante el CVPD.

Algunos motores y gestores optan por establecer dependencias con plataformas externas (de terceros) como una alternativa a crear y proveer una plataforma propia de despliegue y gestión de sistemas. Por ejemplo, motores como Pegasus o DagOnStar delegan las funciones de despliegue de los sistemas de aplicaciones creadas con sus estructuras de procesamiento a plataformas para la gestión de recursos como HTCondor y Slurm, las cuales se encargan de realizar no solo el despliegue de sistemas de aplicaciones, sino también de los recursos que estos utilicen en tiempo de ejecución. Esta restricción establece la instalación de estas plataformas de gestión de recursos, lo cual crea una dependencia funcional entre estas plataformas y los sistemas de aplicaciones. Esta dependencia puede producir interrupción del servicio de procesamiento de los sistemas de aplicaciones para CVPD en escenarios

donde las plataformas de gestión no se encuentren disponibles, o donde surjan retrasos debido a la comunicación entre los sistemas de aplicaciones y dichas plataformas. Este prerrequisito implica que los sistemas de aplicaciones deben crear una configuración de conexión ad hoc con estas plataformas para que estos sistemas funcionen correctamente. Esto produce una dependencia funcional de los sistemas de aplicaciones con las herramientas de gestión de recursos computacionales y cargas de trabajo. Lo cual significa que las aplicaciones de cada etapa dependen de la calendarización y el despacho de datos, los cuales son establecidos por dichas herramientas. Lo anterior, pone a los sistemas en riesgo de acumular retrasos en la generación de productos, así como de crear tiempos ociosos en el flujo de datos y procesamiento. Esto podría llegar a ser crítico en escenarios de toma decisiones sensibles a los tiempos de respuesta.

La dependencia los sistemas de aplicaciones para CVPD con la infraestructura surge cuando su correcta ejecución depende de detalles de una infraestructura específica. Esta dependencia surge porque, aunque los flujos de trabajo y las tuberías son esquemas de procesamiento maduras y ampliamente estudiadas, estas generalmente son desplegadas sobre infraestructuras controladas o semi-controladas, como lo son los clústeres de alto rendimiento o servidores locales [131]. Esta dependencia no crea problemas críticos a los sistemas de aplicaciones cuando se puede crear un contexto computacional conocido antes de desplegar y ejecutar sus aplicaciones.

Sin embargo, recientemente, las organizaciones están utilizando la nube no solo para almacenar, gestionar y procesar los datos producidos por los usuarios asociados a estas organizaciones, sino que también usan este modelo de subcontratación para crear o usar aplicaciones para producir activos y productos digitales [92, 170]. Debido a ello, las organizaciones terminan desplegando múltiples aplicaciones como servicios en la nube para manejar el CVPD [55, 216]. Esto se realiza desde la adquisición de los datos, hasta el procesamiento de estos o la creación de productos digitales. Adicionalmente, y para que sus usuarios puedan acceder a dichos productos, las organizaciones terminan contratando servicios de almacenamiento [188] y logística de entrega/recepción [34, 160]

para gestionar las diferentes versiones de los datos y/o productos digitales que se crean durante un ciclo de vida de los mismos.

De este modo, delegar servicios a los proveedores de servicios de nube permite a las organizaciones reducir costos [98, 218]. Sin embargo, esto también delega el control de estos servicios a los proveedores [151, 153], produciendo una dependencia entre las organizaciones y los proveedores de servicio en la nube (vendor lock-in en inglés) [151, 153]. Esta dependencia podría evitar que las organizaciones puedan crear un ambiente de contexto computacional conocido, lo cual podría crear una dependencia funcional con una infraestructura conocida. Además, podría tener implicaciones críticas de tipo legal en escenarios en los cuales se requiere conocer el contexto de ejecución de las aplicaciones y el manejo, en todo momento, de los productos digitales a través su CV (ciclo de vida). Esto resulta particularmente crítico para solventar leyes asociadas al manejo de datos (p. ej. datos médicos, información industrial o información privada [49, 93, 109]) tomando en cuenta que el proveedor de servicios es honesto pero curioso [2, 63].

Es por esto que, para poder asumir un ambiente de contexto computacional conocido, incluso en la nube, se han creado modelos mediante los cuales las organizaciones pagan por infraestructura privada y de uso exclusivo [7, 109], lo cual, sin embargo, solo está al alcance de grandes corporaciones o gobiernos por su elevado costo y compleja implementación [220, 228]. Incluso cuando se opte por estas soluciones, se podría crear una dependencia a largo plazo entre las organizaciones y el proveedor de servicio debido a los efectos de acumulación de datos en posesión de un solo proveedor, lo cual haría que un potencial cambio de proveedor resulte prohibitivo en términos de costos y tiempo [101, 153, 187] de los procesos de migración entre proveedores de servicios.

En escenarios de migración, se puede presentar una interrupción del procesamiento del sistema de aplicaciones debido a que probablemente esta estructura no funcionará de la misma forma que antes de realizar los cambios de infraestructura [101]. Esto suele suceder porque los desarrolladores acostumbran a utilizar herramientas propias del proveedor en la nube para desarrollar y manejar sus

aplicaciones. Por ejemplo, herramientas comúnmente utilizadas para la creación de flujos de trabajo como Makeflow y Pegasus requieren la instalación de servicios de terceros para la gestión de recursos (p. ej. Slurm [100], HTCondor [69]) y para el manejo de datos (e.g. Amazon WS [10]). En cuyo caso un cambio de proveedor también implicaría un cambio de este tipo de codificación.

Recientemente, se han comenzado a estudiar esquemas para el despliegue de sistemas de aplicaciones para CVPD sobre múltiples infraestructuras como una alternativa para evitar los efectos de dependencia con el proveedor (vendor lock-in). Algunos ejemplos de estos escenarios son la ciencia de datos [209], el big data [11, 53] y el procesamiento en tiempo real (*stream processing* en inglés) [136, 179], lo cual representa una aproximación a soluciones más complejas tales como el agnosticismo² computacional.

1.3 Planteamiento del problema

Como se puede ver, la creación, uso y manejo de estructuras de procesamiento de grandes volúmenes de datos para gestionar ciclos de vida de productos digitales representa un desafío para las organizaciones. Tres vertientes de este desafío se han podido identificar a la luz de los métodos disponibles en el estado del arte y tecnología actualmente usada en la industria:

- La primera se presenta en tiempo de diseño, y es la gestión de la heterogeneidad de las estructuras de procesamiento de datos creadas para manejar un CVPD. Las estructuras de diseño deberían considerar la integración de múltiples aplicaciones escritas en diferentes lenguajes, requiriendo múltiples servicios de manejo de datos desplegados sobre múltiples tipos de plataformas e infraestructuras. Esto se vuelve un desafío cuando se pretende que el diseño

²En inglés, el diccionario de Cambridge define el término agnosticismo como *la creencia de alguien que no conoce, o cree que es imposible conocer si un dios existe* [61]. Específicamente, el agnosticismo es una doctrina filosófica que descarta la noción de absoluto y, especialmente, todo aquello que no puede ser experimentado o demostrado por la ciencia.

posea propiedades genéricas, lo cual permitiría absorber la heterogeneidad de las aplicaciones y su acoplamiento para crear múltiples diseños de estructuras de procesamiento.

- La segunda se presenta en tiempo de despliegue, y son las potenciales dependencias del sistema de aplicaciones con software, plataforma y/o infraestructura que aparecen cuando el control total o parcial del manejo de datos y/o tareas de las estructuras de procesamiento ha sido delegadas a una plataforma externa (delegación de control parcial), o total a un proveedor de servicio (p. ej. nubes privadas, públicas o clústeres de alto rendimiento). Esto se vuelve un desafío, ya que las estructuras de procesamiento dependerían parcial o totalmente del manejo de tareas y datos, lo cual condicionaría la eficacia en la creación de productos digitales a la disponibilidad y eficiencia de los servicios auxiliares. En tal caso, por ejemplo, si la plataforma o la infraestructura fallan o no se encuentran disponibles, las estructuras podrían sufrir cualesquiera de los siguientes incidentes: *i*) no poder completar el despliegue de un sistema de aplicaciones, *ii*) no poder ejecutar un sistema previamente desplegado, *iii*) una aplicación en ejecución podría no encontrar datos de entrada, *iv*) no saber qué aplicación debe recibir sus resultados, o *v*) encontrar un lugar disponible para cargar sus resultados. Todos estos escenarios interrumpirían el CVPD y detendrían la entrega de productos a los procesos de toma de decisión.
- La tercera se presenta en tiempo de ejecución, y se refiere a los potenciales cuellos de botella que se pueden crear cuando las estructuras de procesamiento, desplegadas sobre infraestructuras desconocidas (e.g. nubes o servidores externos), comienzan a crear flujos de datos para procesar cargas de trabajo que no son conocidas previamente y que son atendidas por aplicaciones (etapas del CVPD) cuyo rendimiento no ha sido caracterizado para tal escenario, lo cual es común en el cómputo en la nube. En tal caso, cuando se procesan grandes volúmenes de datos, aparecen escenarios de saturación cuyo comportamiento exponencial depende de la cantidad de etapas del CVPD, el tamaño de los productos digitales y las operaciones de orquestación

de datos.

Las siguientes preguntas de investigación se plantearon a partir de la problemática identificada:

- ¿Cómo gestionar la heterogeneidad en el diseño de las estructuras de procesamiento utilizadas para crear sistemas de aplicaciones para la gestión de CVPD?
- ¿Cómo construir estructuras de procesamiento que permitan reducir, en tiempo de despliegue, las potenciales dependencias de las estructuras de procesamiento con el lenguaje de programación, la plataforma o la infraestructura?
- ¿Cómo proveer a las estructuras de procesamiento con mecanismos que permitan reducir los cuellos de botella que se puedan crear en tiempo de ejecución para mitigar los efectos de escenarios de saturación?

1.3.1 Premisas y formalización de conceptos

En este trabajo de tesis se propone estudiar la propiedad de agnosticismo computacional para dar respuesta a las preguntas recién formuladas. En la literatura, y en términos computacionales, este concepto ha sido utilizado para describir una aplicación, sistema o servicio que, por diseño, no da por sentado que sus requerimientos para funcionar correctamente o mantenerse funcionando se cumplirán de facto en tiempo de despliegue [45].

Lo anterior significa que, para que un componente de software sea considerado agnóstico, este debería estar diseñado para adaptar, en tiempo de ejecución, sus parámetros de contexto (configuración) a los recursos disponibles en la infraestructura en lugar de que estos sean definidos en tiempo de despliegue, como se hace tradicionalmente. Esto permite que los componentes agnósticos (aplicación/sistema/servicio) puedan ser desplegados en múltiples infraestructuras [68, 102, 132, 194].

Especialmente, en la industria el término *agnóstico de la nube* (*cloud agnostic* en inglés) se refiere a una estrategia de diseño para evitar que las aplicaciones, servicios y herramientas se encuentren ligadas funcionalmente a una sola plataforma en la nube y, por lo tanto, este software puede ser desplegado entre diferentes plataformas en la nube y recursos locales [68, 102].

Partiendo de esta definición, el término agnóstico de la infraestructura puede ser definido como la propiedad de una aplicación o servicio para ser desplegado y ejecutada exitosamente, aun cuando se desconozcan los detalles y características de los recursos de la infraestructura y/o la carga de datos a procesar. Esto significa que estas aplicaciones deben identificar las limitaciones y capacidades de la infraestructura en tiempo de ejecución. Esta característica también es conocida como agnosticismo de la infraestructura.

En este contexto, la propiedad de agnosticismo de la infraestructura es conseguida cuando un usuario puede crear, a alto nivel, estructuras de procesamiento de diferentes tipos (no solo tuberías) usando múltiples aplicaciones desarrolladas utilizando diferentes tipos de lenguajes de programación y, en forma transparente y automática, estas estructuras se desplieguen sobre múltiples infraestructuras evitando dependencias (software, plataforma e infraestructura) para finalmente mantenerse procesando datos incluso en escenarios de sobrecargas de trabajo producidos por cargas de trabajo y rendimiento de recursos computacionales desconocidos.

1.4 Hipótesis

A partir de las preguntas de investigación, la hipótesis de esta tesis es la siguiente:

La aplicación, en tiempo de diseño, de un modelo de composición de servicios basado en estructuras autocontenidas y autosimilares en conjunción con la utilización, en tiempo de despliegue, de modelos de programación e infraestructura como código, así como la gestión, en tiempo de ejecución, de cuellos de botella mediante paralelismo implícito producirá estructuras agnósticas de

la infraestructura.

1.5 Objetivo general y objetivos específicos

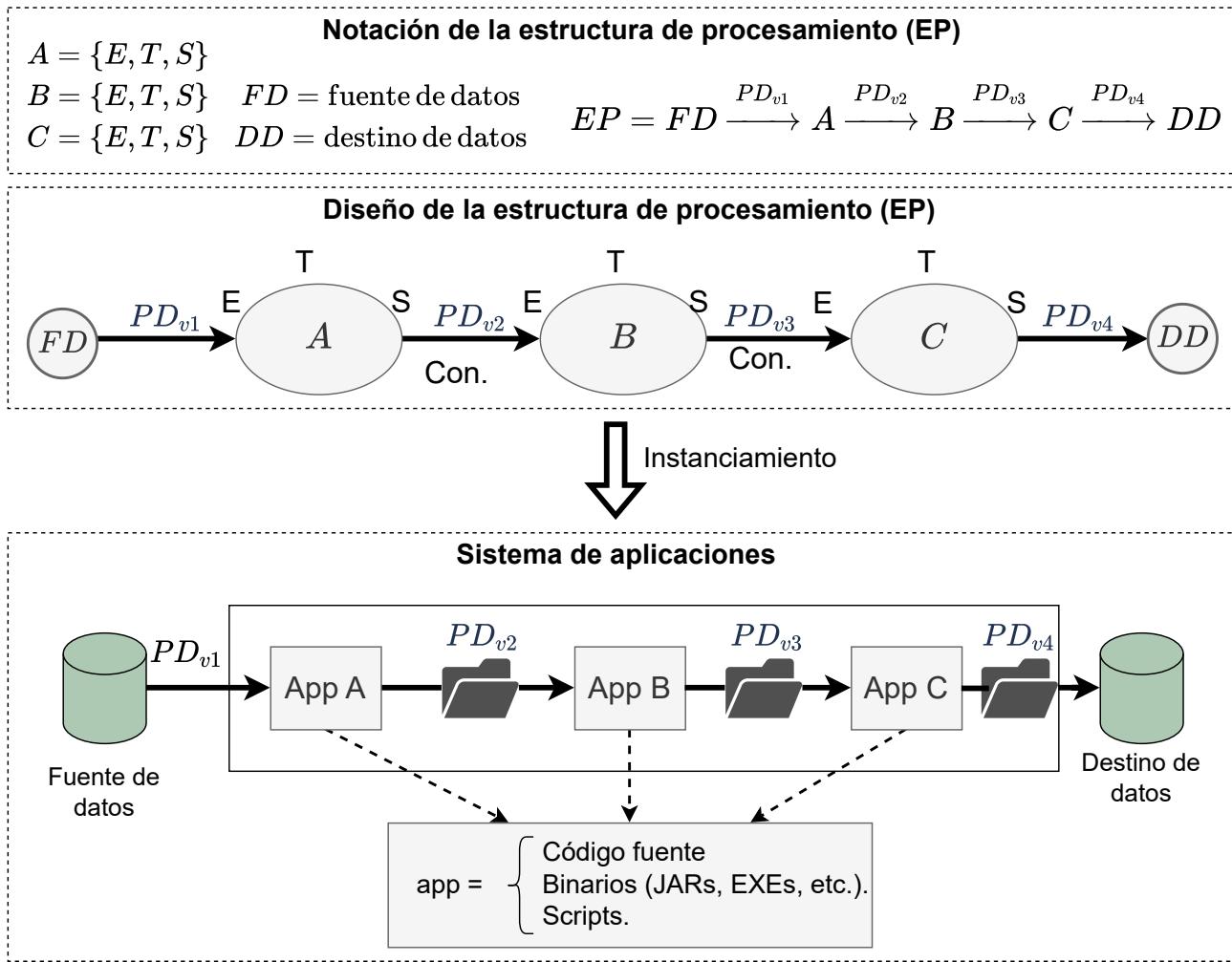
Para abordar la problemática y dar respuesta a la hipótesis de investigación se ha establecido un objetivo general y tres objetivos específicos.

1.5.1 Objetivo general

Crear un método de composición de servicios para diseñar, desplegar y gestionar la operación continua de estructuras de procesamiento agnósticas de la infraestructura.

1.5.2 Objetivos específicos

1. Crear un modelo de construcción basado en bloques autosimilares y autocontenidos para gestionar la heterogeneidad de las estructuras de procesamiento utilizadas para el manejo del ciclo de vida de los productos digitales.
2. Definir y crear un modelo de comunicación y programación de infraestructuras como código para materializar y desplegar, en forma transparente y automática, estructuras de procesamiento sobre múltiples infraestructuras.
3. Obtener un modelo de gestión de tareas/datos y paralelismo implícito para mitigar, en tiempo de ejecución, los efectos de cuellos de botella y evitar la interrupción de procesamiento en escenarios de saturación en las estructuras de procesamiento.



S: Salidas E: Entradas
 Con = Conector T: Transformación PD: Producto digital

Figura 1.6: Representación conceptual del proceso de materialización de una estructura de procesamiento abstracta, creada en tiempo de diseño, en un sistema de aplicaciones en tiempo de despliegue y ejecución.

1.6 Panorama general del método para construcción de estructuras agnósticas

El desarrollo de un sistema de aplicaciones para la gestión de CVPDs incluye la definición de una estructura de procesamiento en donde, abstractamente, se describen en forma inequívoca las reglas y secuencia de ejecución de los futuros componentes de un sistema, lo cual produce como resultado *un patrón, un flujo o una tubería* expresado mediante una fórmula (ver parte superior de Figura 1.6).

La Figura 1.6 muestra una representación conceptual de una estructura abstracta que, en tiempo de despliegue, se materializa en un sistema mediante un proceso llamado *instanciamiento*. Este proceso consiste en el despliegue de un conjunto de componentes siguiendo las reglas definidas en la estructura de procesamiento. La estructura de procesamiento se materializa cuando se asignan valores a la fórmula de dicha estructura (ver parte superior de Figura 1.6). Por ejemplo, las aplicaciones (etapas o nodos) son asignados a binarios de aplicaciones y las entradas/salidas (aristas) son asignadas a rutas en un contexto computacional (ver parte baja de Figura 1.6). Este sistema hereda las reglas de secuenciación de procesos/tareas, así como la adquisición y entrega de datos descritos en la estructura de procesamiento usada como base de diseño. Este proceso convierte una estructura de procesamiento en un sistema que, en tiempo de ejecución, se convierte en una pieza de software que ya puede usar un humano o un robot (otra aplicación).

En este trabajo de tesis se propone agregar la propiedad de agnosticismo a las estructuras de procesamiento con el fin de que estas puedan crear sistemas de aplicaciones para la gestión del CVPD que hagan frente a las diferentes aristas de la problemática previamente identificadas y descritas.

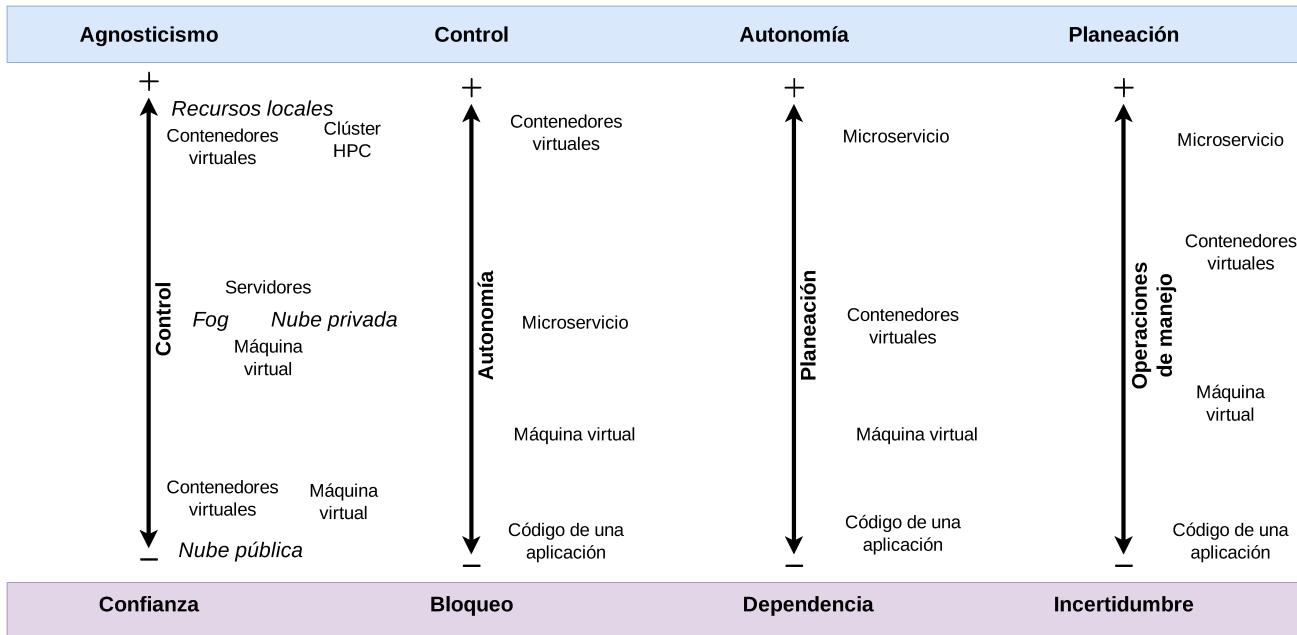


Figura 1.7: Clasificación de soluciones de acuerdo a su nivel de agnosticismo, control, autonomía, planeación y manejo operacional.

1.6.1 Agnosticismo en las estructuras de procesamiento

En la literatura, ya han sido reportados algunos marcos de trabajo para crear aplicaciones agnósticas de la infraestructura [45]. Las aplicaciones desarrolladas usando estos marcos de trabajo pueden ser ejecutadas en diferentes infraestructuras (p. ej. la nube, un clúster, en el grid o en plataformas de contenedores) sin modificar el código fuente de estas aplicaciones, lo cual permite la libre portabilidad de dichas aplicaciones. Sin embargo, estas aproximaciones solo se enfocan en brindar agnosticismo a nivel de aplicación, siendo la construcción de estructuras agnósticas un desafío de investigación abierto.

Para proporcionar un panorama general sobre los requerimientos y acciones requeridas para conseguir agnosticismo durante el ciclo de vida completo de las estructuras de procesamiento (*CVEP*), en la Figura 1.7 se presenta un esquema descriptivo basado en acciones y componentes implicados en el proceso de diseño, despliegue y ejecución de un *CVEP*. En la Figura 1.7 se muestran los diferentes

aspectos requeridos para crear una solución agnóstica. Partiendo de derecha a izquierda, se pueden observar las siguientes características para lograr una solución agnóstica: manejo de operaciones, planeación, autonomía y control. El manejo de operaciones se refiere a las estrategias que un diseñador incluye en el software para mitigar de forma preventiva, correctiva y reactiva los diferentes problemas de dependencias funcionales que pueden surgir en la estructura de procesamiento durante tiempos de diseño, despliegue y ejecución, respectivamente. Por lo tanto, mientras menos operaciones de manejo delega el diseñador, incrementa el grado de *planeación*, permitiendo que ellos manejen el comportamiento de su estructura durante tiempos de diseño, despliegue y ejecución.

Un mayor grado de planeación incrementa la autonomía de las estructuras de procesamiento, sin delegar la gestión de estas a un tercero (proveedor de servicios en la nube).

Cuando la autonomía de las soluciones aumenta, los diseñadores ganan control sobre su software y sobre como este es manejado en la infraestructura y plataforma, disminuyendo los riesgos de generar un bloqueo con un proveedor (*vendor lock-in*). Por ejemplo, soluciones autónomas como los contendores virtuales y los microservicios permiten a los diseñadores tener el control sobre los mecanismos de comunicación y escalamiento de las aplicaciones, mientras que en tecnologías como las plataformas FaaS (function-as-a-service) los diseñadores pierden este control, y el control de las aplicaciones está bloqueado a los servicios/características especificadas por el proveedor del servicio. En este sentido, un grado alto de control produce una solución agnóstica que puede ser desplegada en múltiples plataformas e infraestructuras (p. ej. recursos locales o la nube pública y privada). Mientras que, al disminuir el *control* sobre el software, plataforma e infraestructura, los diseñadores *confían* en los mecanismos implementados por el proveedor de servicios.

Para agregar la propiedad de agnosticismo a los componentes de una estructura de procesamiento, se pueden identificar tres retos:

1. Las estructuras de procesamiento deberían ser construidas a alto nivel y, mediante procesos declarativos, permitir a los diseñadores no depender de los lenguajes de programación usados

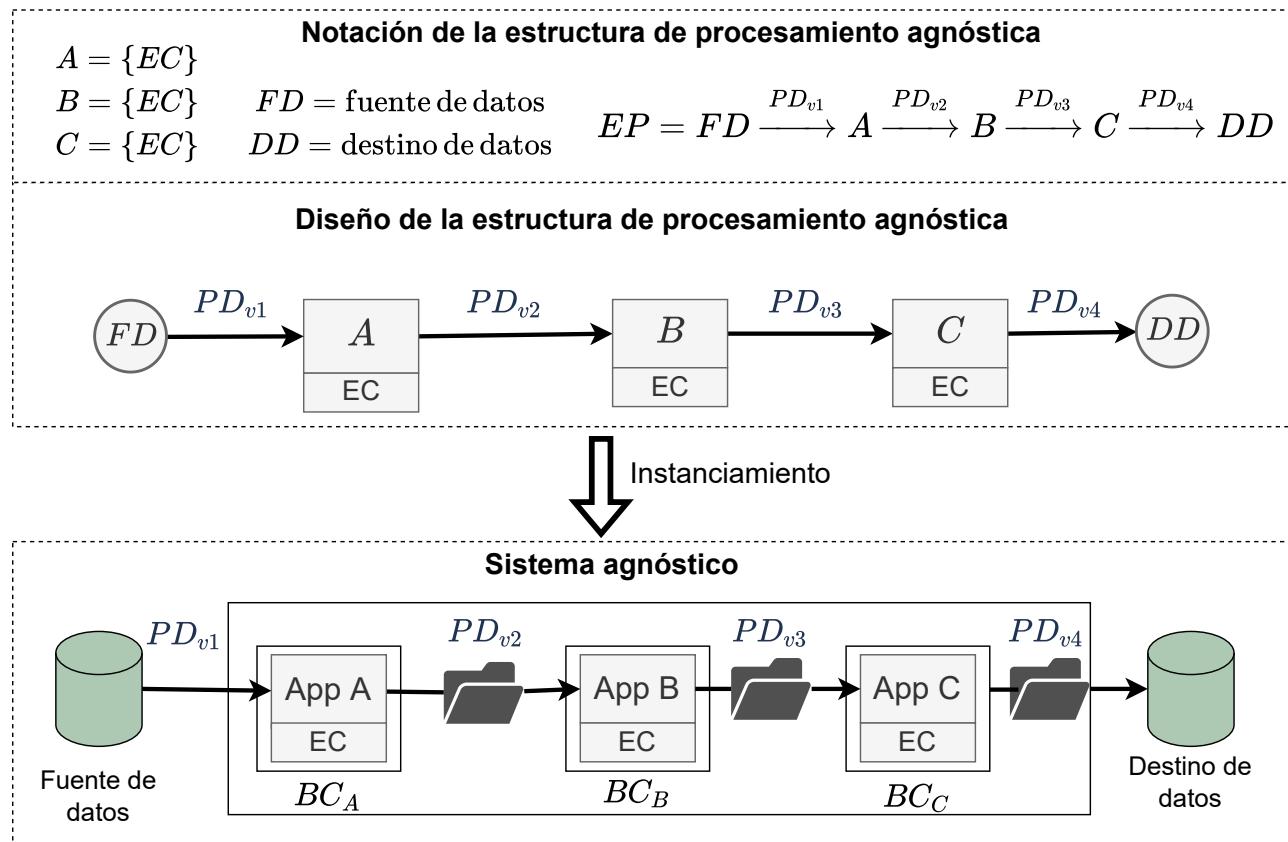
por desarrolladores o de los tipos de estructuras (p. ej. tuberías o patrones) disponibles en un marco de trabajo (frameworks).

2. Las estructuras diseñadas a alto nivel deberían, en tiempo de despliegue, materializarse automáticamente sin generar dependencias entre el diseño de la estructura de procesamiento y la plataforma usada para su despliegue (e.g. que una tubería solo funcione en un ambiente específico de nube pública). Esto implica que las estructuras deberían poder manejar flujos de datos a través de múltiples plataformas e infraestructuras.
3. Una vez instanciadas, las estructuras de procesamiento deberían, en tiempo de ejecución, realizar el manejo implícito de las colas de cada etapa de las estructuras de procesamiento para evitar la interrupción del procesamiento en escenarios de saturación.

Esto significa que un método cuyo objetivo sea dotar a las estructuras de procesamiento con la propiedad de agnosticismo de la infraestructura debe considerar el *CVEP*. En este ciclo de vida se consideran etapas tradicionales tales como diseño (creación y codificación), despliegue y ejecución.

La Figura 1.8 muestra el proceso para crear un sistema de aplicaciones a partir del diseño de una estructura de procesamiento agnóstico. La idea básica es crear un método de construcción que permita diseñar estructuras de procesamiento que incluyan, embebidos en sus nodos, un conjunto de mecanismos creados mediante la implementación de modelos de comunicación y programación para conseguir agnosticismo en tiempo de despliegue, y modelos reactivos (parallelismo e identificación de cuellos de botella) para prevenir escenarios de saturación en tiempo de ejecución (ver estructuras de control *EC* en la Figura 1.8).

Tomando en cuenta que las estructuras de procesamiento regulan el comportamiento de los sistemas de aplicaciones, se espera que estos sistemas hereden el comportamiento agnóstico. El resultado sería un esqueleto (con propiedades agnósticas) sobre el cual, automáticamente, se deposite el sistema de aplicaciones de gestión del CVPD. Por ejemplo, el sistema de aplicaciones conformado por App



EC: Estructuras de control

BC: Bloque de construcción

PD: Producto digital

Figura 1.8: Representación conceptual del proceso de convertir el diseño de una estructura de procesamiento agnóstica en un sistema agnóstico de aplicaciones.

A, App B y App C en la Figura 1.8, las cuales están montadas sobre las estructuras de control y el esqueleto expresado como un conjunto de bloques de construcción (BC_A , BC_B y BC_C).

La meta de este método es permitir a los participantes asociados de las diferentes etapas del *CVEP* permanecer *agnósticos* sobre las tareas que desarrollarán otros participantes. Por ejemplo, los desarrolladores solo confiarán en el código de sus aplicaciones, para lo cual solo requerirían declarar requerimientos (p. ej. dependencias, librerías o sistemas operativos) de su código. Los arquitectos solo confiarán en su diseño estructural (p. ej. tuberías, flujos, patrones, etc.) y solo tendrán que declarar los métodos y modelos de acceso a los recursos de cada etapa de la estructura diseñada (estructuras de control habilitadas para cada esqueleto). Mientras que, los administradores de tecnologías de

información (TIC) solo deberán confiar en detalles contextuales con el fin de proveer correctamente datos contextuales (p. ej. rutas de entrada, salida o binarios, así como requerimientos no funcionales para cada caso). Finalmente, los proveedores de servicio o administradores de TIC solo deberán confiar en proporcionar la infraestructura solicitada y no se ocuparán de la gestión de las estructuras en tiempo de ejecución, ya que esto, en un escenario agnóstico, debería ser gestionado por las estructuras de control del esqueleto (ECs y BCs).

1.6.2 Metodología para crear estructuras de procesamiento agnósticas de la infraestructura

En la presente tesis, se propone un método que permite el diseño, despliegue y ejecución de estructuras de procesamiento agnósticas de la infraestructura, gestionando la heterogeneidad de las estructuras de procesamiento durante tiempo de diseño, las dependencias que pueden surgir durante tiempo de despliegue y la operación continua durante tiempo de ejecución.

En este sentido, el método propuesto está conformado por tres modelos (uno por objetivo), los cuales se ilustran en la Figura 1.9: *i)* un modelo de construcción basado en bloques autosimilares y autocontenidos, *ii)* un modelo de comunicación y programación de infraestructuras como código, y *iii)* un modelo de gestión de tareas/datos basado en paralelismo implícito.

El modelo de construcción permitirá encapsular las aplicaciones consideradas en una estructura de procesamiento para manejar un CVPD en bloques autosimilares y autocontenidos. Estos bloques permiten abstraer y gestionar la heterogeneidad de estas aplicaciones al encapsularlas en un contenedor virtual con sus dependencias y un conjunto de estructuras de control, las cuales permiten la interconexión de las aplicaciones aun cuando estas se encuentren desarrolladas en diferentes lenguajes de programación, mitigando la dependencia con el lenguaje de programación en tiempo de diseño. Este modelo de construcción permite a los diseñadores crear diferentes patrones (p. ej.

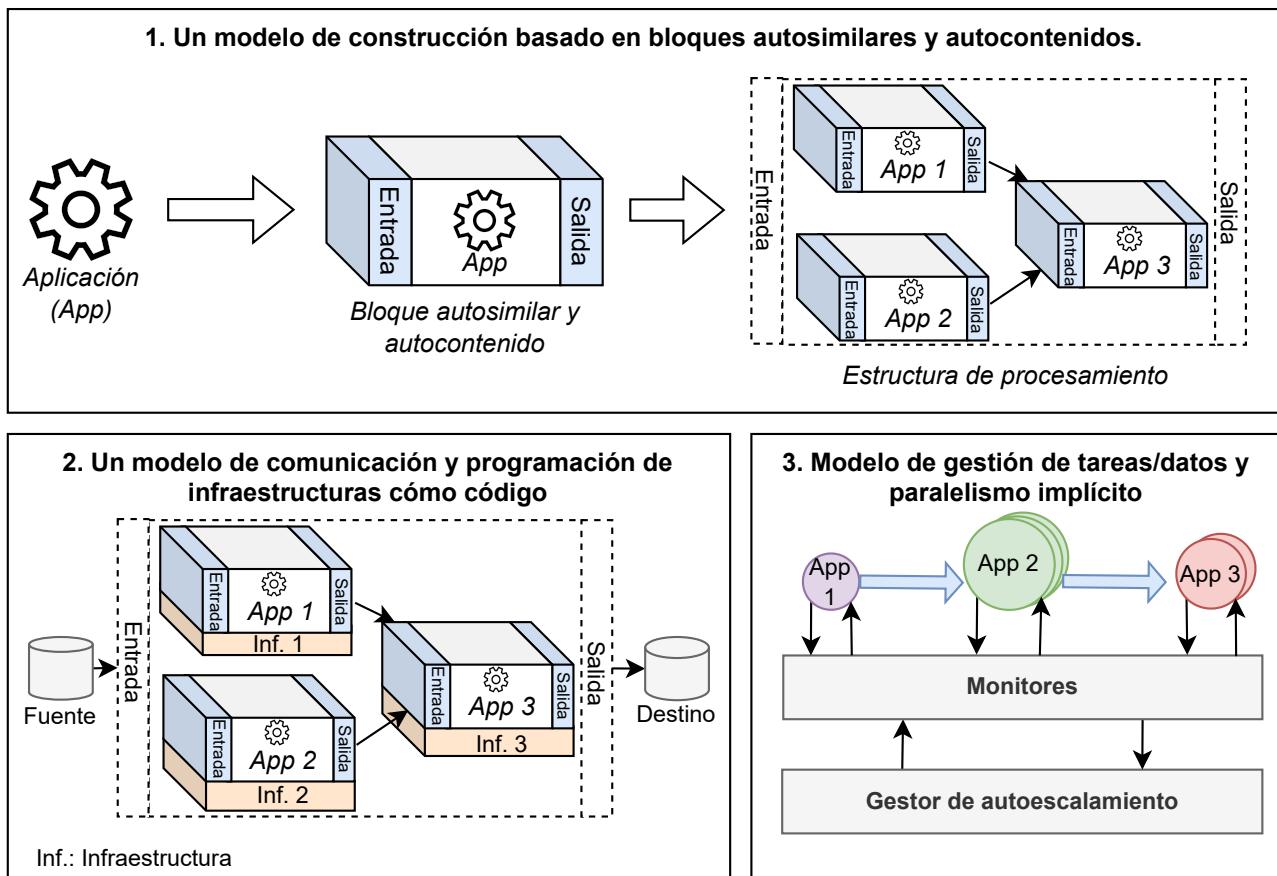


Figura 1.9: Diseño conceptual de la metodología seguida en esta tesis para crear estructuras de procesamiento agnósticas de la infraestructura.

tuberías o flujos de trabajo) para crear sus estructuras de procesamiento.

Una estructura construida con bloques autocontenidos es aquella en la cual cada uno de sus módulos se desarrolla para poder trabajar de forma independiente al resto de bloques. Esto quiere decir que cada bloque puede ser remplazado por otro, sin afectar al resto de componentes en una estructura. Lo anterior, se consigue al agregar una arquitectura de microservicios a cada bloque para convertirlo en una estructura autocontenido (o microservicio). Estos bloques tienen una funcionalidad definida por la aplicación encapsulada originalmente en el bloque, de tal suerte que este nuevo bloque autocontenido puede ser usado en forma independiente como una pieza de software. Ser “autocontenido” significa que al bloque original se le han agregado todos los microcomponentes (dependencias, librerías,

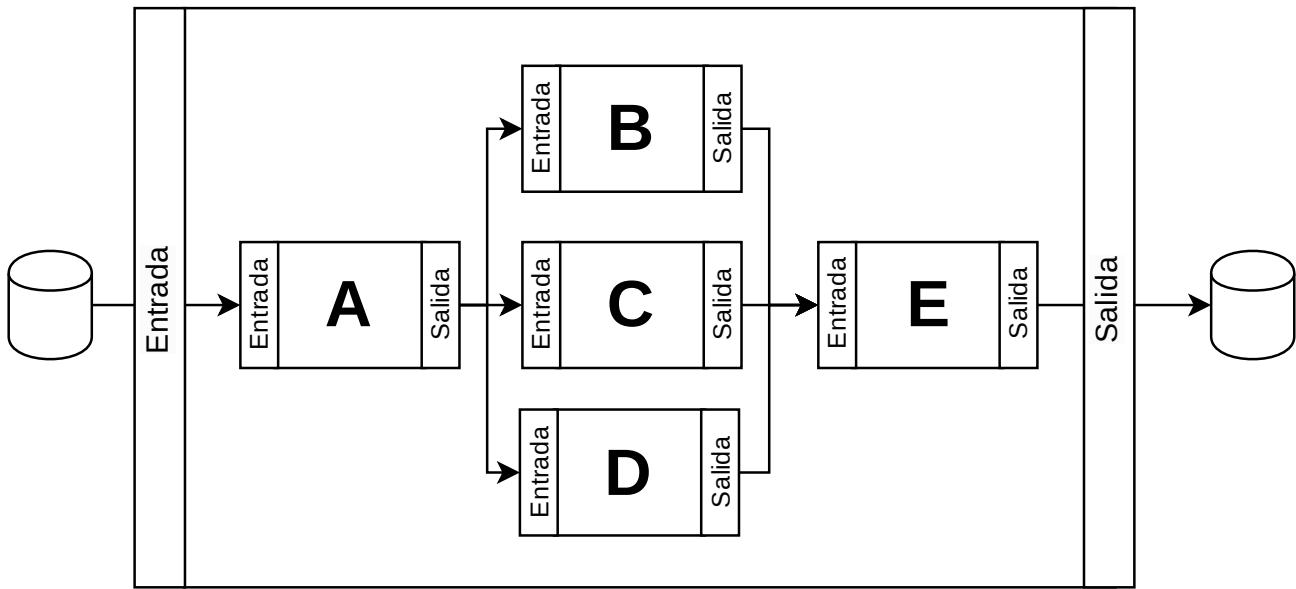


Figura 1.10: Ejemplo conceptual de una estructura autosimilar.

variables de ambiente, modelos de comunicación y autogestión) requeridos para funcionar por sí mismo y de esta forma evitar depender del resto de microservicios.

Por otra parte, un bloque autosimilar es aquel en el que su diseño es conceptualmente similar al del resto de componentes, y que al unirse con otros forman una estructura de procesamiento que sigue el mismo principio de diseño que sus componentes. Por ejemplo, en la Figura 1.10 se observa como cada componente (A, B, C y D) tienen un diseño de tres capas verticales (entrada, procesamiento y salida). Al unir estos componentes, se forma una estructura de procesamiento que, conceptualmente, es similar a estos, dado que también cuenta con capas de entrada, procesamiento y salida. Este concepto de diseño de estructuras autosimilares permite el manejo a diferentes niveles de composición de sistemas y servicios utilizados en el manejo del CVPD.

En tiempo de despliegue, un modelo de comunicación y programación de infraestructuras como código (IaC, por su definición en inglés) materializa y despliega, en forma transparente y automática, estructuras de procesamiento sobre múltiples infraestructuras. Este modelo permite que se mitigue la dependencia de los sistemas de aplicaciones con la plataforma y la infraestructura al no delegar

la orquestación de datos y tareas a herramientas de terceros. Como resultado, las estructuras de procesamiento manejadas con este modelo son dotadas con los requerimientos no funcionales de portabilidad y reusabilidad, posibilitando que puedan ser migradas en diversas infraestructuras y plataformas, y que puedan ser reutilizadas en diferentes soluciones para el manejo del CVPD.

Finalmente, en tiempo de ejecución, un modelo de gestión de tareas/datos y paralelismo implícito mitigará los efectos de los cuellos de botella en una estructura de procesamiento, evitando la interrupción del procesamiento continuo de los datos en escenarios de saturación y gestionando la carga de trabajo entrante. En este sentido, este modelo monitorea, en forma implícita, las colas y el rendimiento de las etapas de una estructura de procesamiento para identificar cuellos de botella y mitigarlos utilizando un esquema de autoescalamiento.

1.7 Metodología de investigación

En esta sección, se describe la metodología seguida en este trabajo de investigación para alcanzar los objetivos establecidos, así como para validar la hipótesis de investigación. Esta metodología está dividida en seis etapas, las cuales se ilustran en la Figura 1.11:

Etapa 1 Revisión del estado del arte para identificar modelos, marcos de trabajo y herramientas que aborden la construcción de estructuras de procesamiento para el manejo del ciclo de vida de los productos digitales. Esta etapa incluye el estudio de los retos existentes en el área de investigación de la presente tesis. Además, en esta etapa se establecieron los requerimientos y limitaciones para construir estructuras de procesamiento agnósticas de la infraestructura.

Etapa 2 Tomando en consideración los requerimientos y limitaciones identificados en la *Etapa 1*, en esta etapa se considera el diseño, desarrollo y evaluación de un modelo de construcción, basado en bloques de construcción autosimilares y autocontenidos que permita el diseño de estructuras de procesamiento agnósticas de la infraestructura. El producto de esta fase es un modelo

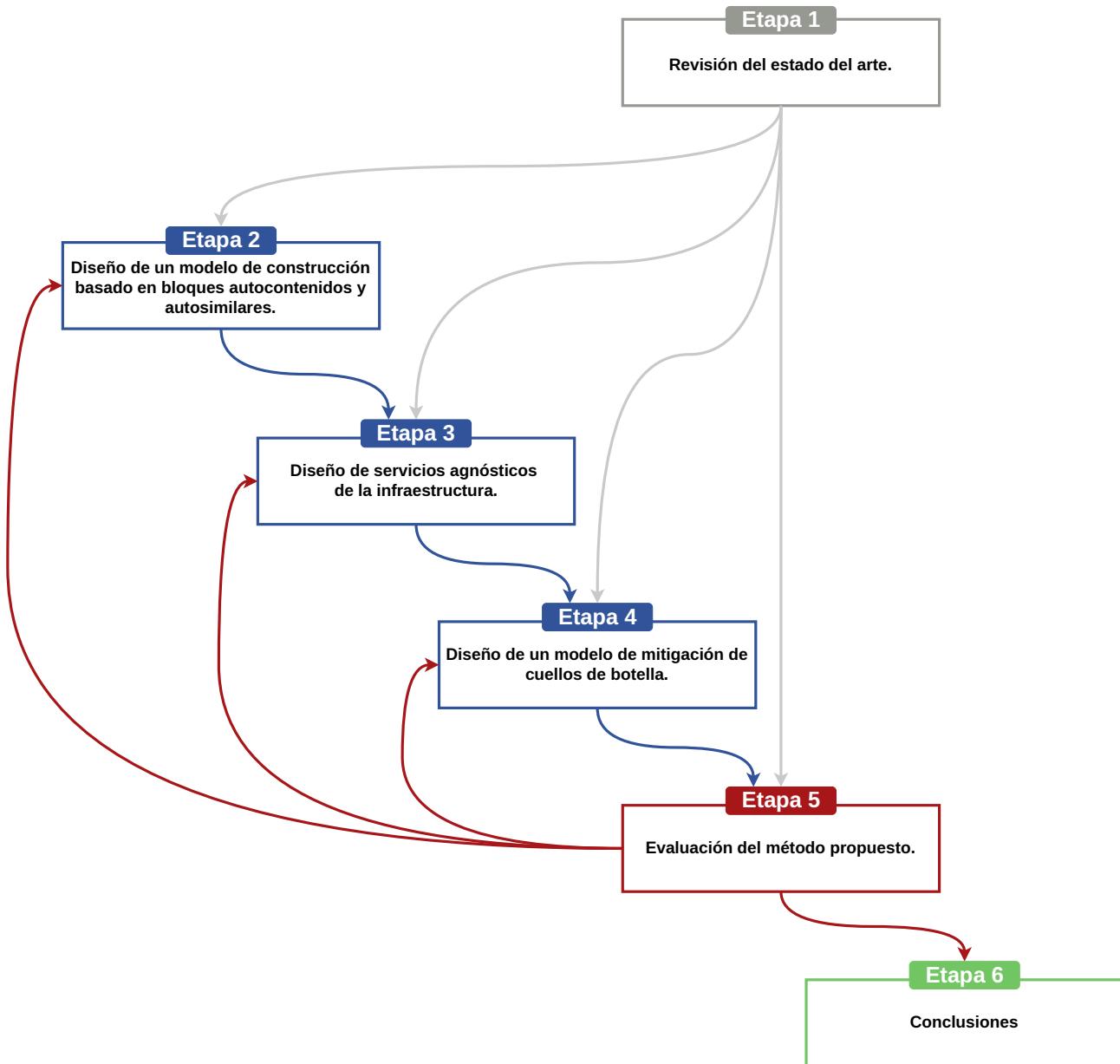


Figura 1.11: Metodología de investigación propuesta.

que permitirá a las organizaciones la encapsulación en bloques de construcción las aplicaciones utilizadas en los ciclos de vida de sus productos digitales. Estos bloques podrán ser desplegados en múltiples infraestructuras y encadenados entre sí para formar estructuras de procesamiento que permitan el manejo del ciclo de vida de los productos digitales.

Etapa 3 Partiendo de los bloques de construcción desarrollados en la *Etapa 2*, en esta etapa se agregarán a estos bloques modelos de procesamiento y comunicación para permitir la construcción de estructuras de procesamiento agnósticas de la infraestructura, compuestos de bloques de construcción desplegados en múltiples infraestructuras. Como resultado, en esta etapa se crearán estructuras de procesamiento, desplegadas como servicios y construidas con bloques de construcción. El objetivo en esta etapa es habilitar la entrega continua de datos a través de los componentes de la estructura de procesamiento, los cuales son aplicaciones utilizadas en el ciclo de vida de un producto digital y que se encuentran distribuidas en diferentes infraestructuras.

Etapa 4 En esta etapa se identificarán y mitigarán, cuando sea posible, los cuellos de botella que puedan surgir en las estructuras de procesamiento diseñadas en la *Etapa 3*. Para este fin, se diseñarán un modelo de mitigación de cuellos de botella basado en técnicas de manejo de colas. Con la mitigación de los cuellos de botella se espera que el tiempo de respuesta observado por los usuarios se reduzca y, por lo tanto, mejore su experiencia al utilizar las estructuras de procesamiento diseñadas.

Etapa 5 En esta etapa se evaluará el método desarrollado en las etapas anteriores. La experimentación considera la conducción de diferentes estudios de caso basados en el manejo de imágenes satelitales y contenidos médicos. Esta etapa también servirá para identificar áreas de mejora en el método desarrollado.

Etapa 6 En esta etapa se definirán las conclusiones finales obtenidas a partir de los experimentos realizados en la *Etapa 5*, así como del desarrollo durante esta investigación en las diferentes Etapas de la metodología.

1.8 Organización del documento

El resto del documento está organizado de la siguiente manera. El Capítulo 2 presenta el marco teórico, el cual contiene algunos conceptos empleados en esta tesis, entre los que se incluyen el procesamiento de datos a gran escala, big data, patrones de paralelismo y requerimientos de software. En el Capítulo 3 se describe los principales trabajos relacionados con herramientas para el diseño de estructuras de procesamiento, arquitecturas de microservicios, el manejo de requerimientos no funcionales en estructuras de procesamiento y cómputo agnóstico. El Capítulo 4 describe el método de procesamiento continuo propuesto para construir estructuras de procesamiento para el manejo del ciclo de vida de productos digitales. En el Capítulo 5 se presenta la evaluación experimental del método propuesto, conducida en la forma de estudios de caso. Finalmente, en el Capítulo 6 se describen las principales conclusiones, limitaciones y trabajo futuro de la presente tesis.

2

Marco teórico

En este Capítulo se introducen diferentes conceptos y técnicas esenciales para comprender la problemática abordada en la tesis. En la primera parte de este Capítulo, se presenta una breve descripción sobre los conceptos de requerimientos funcionales y no funcionales de software, los cuales son claves en el diseño de una estructura de procesamiento. En la siguiente parte de este Capítulo se describen diferentes arquitecturas de software y patrones de diseño que se pueden seguir durante el diseño de una estructura de procesamiento. Posteriormente, se hace una introducción a los motores de flujos de trabajo, los cuales comúnmente son utilizados en el desarrollo de flujos de trabajo, los cuales son estructuras de procesamiento utilizadas para la automatización de un conjunto de tareas repetitivas. En este Capítulo, además, se presenta el concepto de tubería de datos, las cuales son diseñadas para realizar el procesamiento de grandes volúmenes de datos en problemas de big data (procesamiento por lotes y en tiempo real). Finalmente, se describen diferentes infraestructuras comúnmente utilizadas para el despliegue de estructuras de procesamiento, así como los conceptos

de contenedores virtuales y cómputo sin servidor, los cuales son estrategias de despliegue y manejo de software que se han popularizado en años recientes.

2.1 Requerimientos de software funcionales y no funcionales

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, por sus siglas en inglés) en su estándar 610.12-1990 [99], define un requerimiento de software como “una condición o capacidad que debe de ser cumplimentada por un sistema o alguno de sus componentes para satisfacer un contrato, estándar, especificación o cualquier otra imposición establecida en un conjunto de documentos”.

Los requerimientos de software se dividen en requerimientos funcionales y no funcionales. Los requerimientos funcionales son aquellos que definen la funcionalidad y comportamiento de una estructura de los componentes (aplicaciones y etapas) que forman un ciclo de vida. Estos requerimientos deben de ser cumplimentados de forma obligatoria, debido a que son funciones básicas requeridas por los usuarios. De esta manera, un requerimiento funcional puede ser una tarea, acción o actividad requerida en el ciclo de vida [44]. Por ejemplo, en un ciclo de vida para el manejo de datos médicos se pueden considerar requerimientos funcionales tales como la anonimización de imágenes médicas o la visualización de las imágenes.

Por otro lado, los requerimientos no funcionales describen características de calidad que deberían ser cumplimentados en un ciclo de vida. Estos requerimientos no son obligatorios debido a que no definen el comportamiento ni funcionamiento de los componentes del ciclo de vida [44]. Estos requerimientos son agregados en un ciclo de vida para hacer frente a retos de eficiencia, seguridad, confiabilidad y escalabilidad, por nombrar algunos ejemplos. En este sentido, los requerimientos no funcionales se pueden clasificar en cinco grupos de acuerdo a la problemática que abordan: interfaz de usuario (p.

ej. usabilidad), desempeño (p. ej. eficiencia en términos de tiempos de respuesta y almacenamiento), ciclo de vida (p. ej. portabilidad, reusabilidad y mantenibilidad), y económicos (p. ej. costo-eficiencia) [1, 44].

Los siguientes requerimientos no funcionales son considerados en esta tesis para la construcción de estructuras de procesamiento agnósticas de la infraestructura:

Confiabilidad. Permite a un sistema seguir operando aún en casos donde alguno de sus componentes entre en estado de falla. La *confiabilidad sistémica* permite la tolerancia a fallos de un componente del sistema [77], mientras que la *confiabilidad de datos* permite resistir fallas en la infraestructura donde los datos están almacenados [200].

Eficiencia. Desde el punto de vista de procesamiento de datos, se refiere a la capacidad de una solución de utilizar los recursos de hardware y software disponibles para aumentar su desempeño [3]. Por ejemplo, utilizando técnicas de paralelismo para hacer uso de múltiples CPUs para procesar los datos.

Manejo de la heterogeneidad. Permite la interconexión de múltiples aplicaciones desarrolladas en diferentes lenguajes de programación sin tener que modificar su código [145, 174].

Portabilidad. Se refiere a la capacidad de un sistema o aplicación para poder ser desplegado en múltiples infraestructuras y plataformas sin tener que modificar su código [87, 152].

Reusabilidad. Se refiere a la capacidad de un sistema, aplicación o servicio de volver a utilizar alguno de sus componentes para crear una nueva solución. Este requerimiento permite tomar ventaja de aplicaciones, servicios y estructuras de procesamiento previamente probados y desplegados [159].

2.2 Arquitecturas de software

El IEEE define una arquitectura como “*la organización fundamental y los principios que guían el diseño y evolución de un sistema, sus componentes, las relaciones entre estos componentes y su ambiente de ejecución.*” [126]. En este sentido, una arquitectura de software puede ser definida como el modelo teórico que los diseñadores y desarrolladores deben de seguir para organizar su código en una pieza funcional de software. Este modelo incluye la definición de requerimientos funcionales y no funcionales.

2.2.1 Arquitectura de N-capas

En una arquitectura de N-capas, también llamada arquitectura de múltiples niveles, el software es diseñado desacoplando las operaciones físicas y lógicas del procesamiento de datos, manejo de datos y la presentación/visualización de datos [130]. Estas operaciones son desplegadas en múltiples equipos (servidores) sin compartir recursos entre sí. En este contexto, en este modelo se pueden identificar los siguientes roles:

- Módulos: software de propósito específico.
- Capas: conjunto de módulos organizados como una sola unidad.
- Nivel: unidades de despliegue (infraestructura).
- Inquilinos: usuarios concurrentes a los que se les da servicio.

Las principales ventajas de este modelo es el soporte para el manejo de múltiples aplicaciones y el uso de una protocolo o API común para acceder a los servicios. Mientras, que las principales desventajas son los costos que se producen al manejar múltiples equipos, así como la complejidad de manejar los componentes de la arquitectura (sincronización y balanceo de carga) y las fallas que se pueden

producir en la infraestructura [130].

2.2.2 Arquitectura orientada al servicio

En una arquitectura orientada al servicio (SOA, por sus siglas en inglés) los programas y aplicaciones son llamados “servicios” y ofrecen funcionalidades a otros programas y componentes [115]. Un servicio es accesible a través de paso de mensajes utilizando interfaces de entrada y salida, las cuales son desacopladas de la implementación de cada servicio.

Esta arquitectura tiene una serie de ventajas, tales como su flexibilidad para agregar y remover servicios, su modularidad, su fácil mantenimiento y reutilización de componentes, así como que los servicios pueden ser fácilmente escalados creando sistemas distribuidos. Sin embargo, esta arquitectura tiene como desventajas la sobrecarga que se genera al distribuir los servicios y la inversión en tiempo requerida para migrar una aplicación monolítica a este tipo de arquitectura.

2.2.3 Arquitectura de microservicios

Las arquitecturas de microservicios fueron desarrolladas a partir de los conceptos fundamentales de las arquitecturas orientadas a servicios (SOA, por su definición en inglés) [115]. En una arquitectura de microservicios, las aplicaciones son descompuestas en un conjunto de servicios que son independientes entre sí en términos de desarrollo y despliegue. En esta arquitectura se hace un énfasis en el bajo acoplamiento y alta cohesión de sus componentes [64]. Los microservicios son módulos de grano fino, en comparación con los módulos utilizados en SOA, que tienen las siguientes características:

- Tamaño: los microservicios son pequeños en tamaño en comparación con soluciones monolíticas. En este sentido, una aplicación monolítica es dividida en pequeños servicios, lo cual mejora la mantenibilidad y extensibilidad de la aplicación.

- Contexto acotado: las funciones relacionadas de una aplicación son combinadas en un solo microservicio, el cual contiene todo lo requerido para funcionar sin depender del resto de los microservicios.
- Independencia: cada microservicio es operacionalmente independiente del resto de microservicios y servicios. En una arquitectura de microservicios, la comunicación entre componentes se realiza a través de sus interfaces de entrada y salida.
- Flexibilidad: los microservicios pueden ser agregados y remplazados de la arquitectura sin tener que hacer modificaciones en el resto de microservicios.
- Modularidad: los microservicios son componentes aislados que son acoplados para contribuir en el comportamiento de un sistema más grande. Estos microservicios son fácilmente intercambiables para actualizar el comportamiento del sistema.

Una arquitectura de microservicios tiene diferentes ventajas, tales como el manejo de la heterogeneidad en el desarrollo de los componentes de la arquitectura, la resiliencia de sus componentes, la escalabilidad de sus componentes y su fácil despliegue [64, 144]. Sin embargo, esta arquitectura también tiene como principal desventaja que el manejo de los componentes se puede volver compleja cuando la arquitectura se compone de una gran cantidad de microservicios [172].

2.3 Patrones de diseño

En una estructura de procesamiento, los patrones de diseño son estrategias genéricas y recusables implementadas para hacer frente a distintos problemas [127, 155]. Estos patrones hacen que una solución creada para resolver un problema dentro de un contexto específico pueda ser replicada para resolver problemas similares en diferentes contextos [173]. Algunos ejemplos de patrones de diseño

son el llamado patrón de capas, el patrón cliente/servidor, el patrón de tuberías y filtros, patrón maestro/esclavo, patrón broker y patrón a pares, por mencionar algunos [127, 155]. Ejemplos de patrones arquitectónicos son los siguientes:

Patrón de capas. En este patrón el software se organiza siguiendo una estructura de capas interconectadas y jerárquicas. En este sentido, las capas superiores consumen los servicios provistos por las capas inferiores. Comúnmente, este patrón considera tres capas: acceso, procesamiento y datos.

Patrón cliente/servidor. En este patrón, una entidad llamada servidor se encarga de proveer un servicio a múltiples clientes.

Patrón maestro/esclavo. En este patrón un servicio es replicado en múltiples instancias llamadas esclavos, y una entidad llamada maestro se encarga de distribuir las solicitudes entrantes entre estos esclavos.

Patrón de tuberías y filtros. Las aplicaciones se organizan secuencialmente en abstracciones llamadas *filtros*, las cuales son interconectadas utilizando sus interfaces de entrada y salida mediante abstracciones llamadas *tuberías*. En este sentido, los datos se transportan a través de las tuberías y son procesados en cada filtro.

Patrón broker. En este patrón, un conjunto de servicios son coordinados por una entidad llamada broker, la cual es responsable de recibir las solicitudes de un cliente y redireccionarlas al servicio que debe de atender la solicitud.

Patrón a pares. Un conjunto de servicios llamados pares (peers en inglés) son interconectados siguiendo una topología de anillo. Cada par implementa un patrón cliente-servidor para comunicarse con el resto de servicios en el patrón.

2.4 Motores de flujos de trabajo

Un flujo de trabajo de procesamiento es diseñado, desplegado, ejecutado y gestionado por dos componentes: un motor de flujos de trabajo y un sistema de gestión de flujos de trabajo [21, 174]. El motor de flujos de trabajo se encarga del despliegue de los flujos de trabajo, siguiendo un DAG que contiene la declaración de los nodos y dependencias de datos de un flujo de trabajo [21, 201]. Siguiendo la configuración del DAG, los nodos son desplegados en la infraestructura disponible, y el sistema de gestión de flujos de trabajo se encarga de controlar la ejecución de los componentes de los flujos de trabajo durante tiempo de ejecución. Por lo tanto, este gestor se encarga del acoplamiento de los nodos en un flujo de trabajo mediante el uso de sus interfaces de E/S (e.g. FTP, GridFTP, HTTP, SCP, TCP/UDP Sockets), así como de la inyección de datos [120].

En la literatura, los motores de flujos de trabajo comúnmente se encuentran embebidos en el sistema de gestión de flujos de trabajo, el cual se encarga de monitorear y ejecutar los flujos de trabajo de forma transparente, ocultando los detalles de orquestación e integración de las tareas distribuidas [131, 174]. Rodríguez y Buyya [174] presentan una arquitectura general de los sistemas de gestión de flujos de trabajo, donde el motor de flujos de trabajo es el núcleo del sistema y maneja la ejecución de los flujos de trabajo siguiendo un DAG escrito en un lenguaje de alto nivel como Python o en archivos de configuración como XMLs. La Figura 2.1 describe esta arquitectura en donde el resto de los componentes son una interfaz de usuario, herramientas de administración y monitoreo, servicios de información en la nube y APIs de proveedores en la nube.

2.4.1 Paralelismo en flujos de trabajo

Un flujo de trabajo puede ser diseñado para el procesamiento de grandes volúmenes de datos [20, 111]. En este sentido, los sistemas de gestión de flujos de trabajo implementan técnicas de paralelismo que

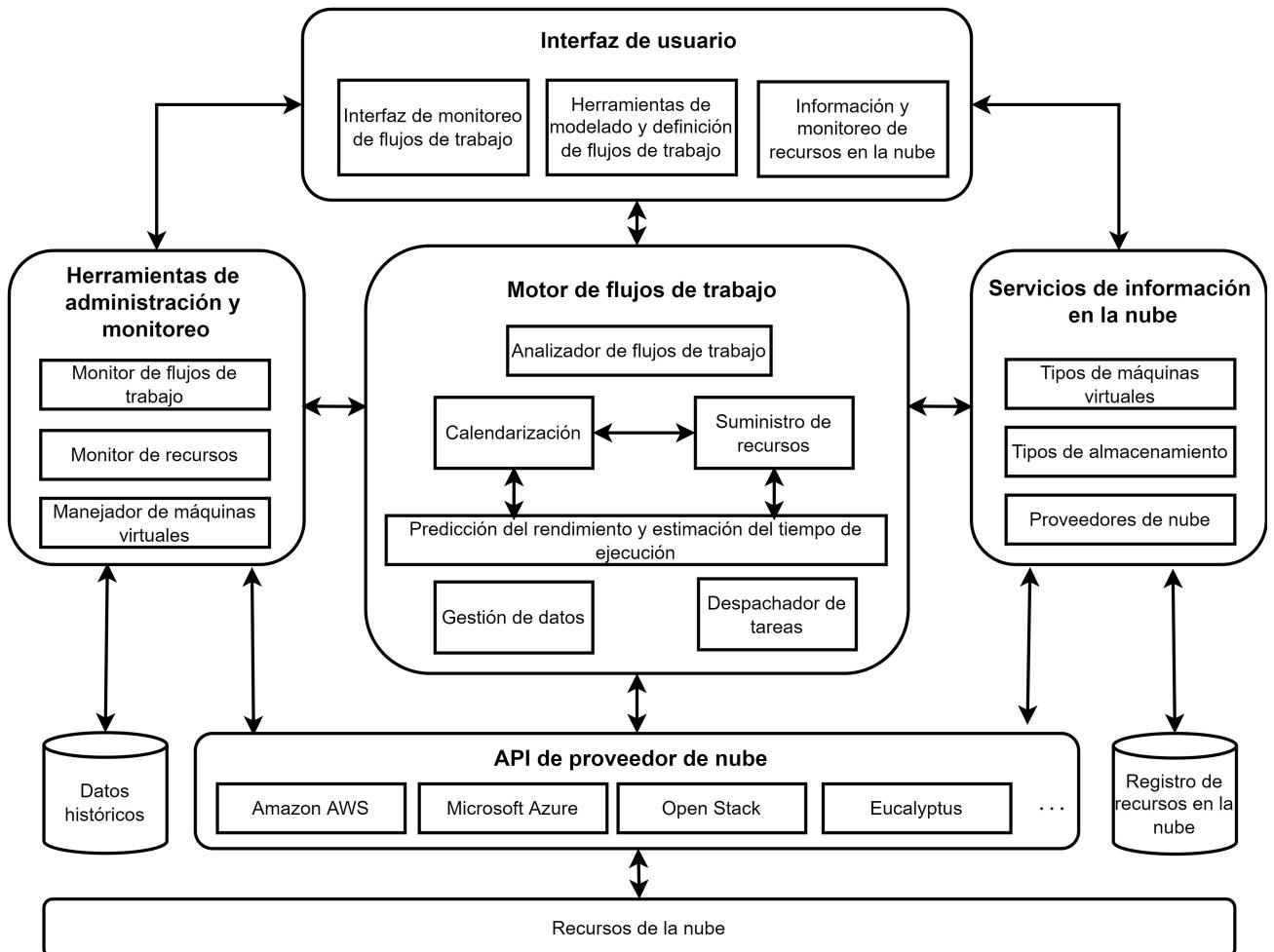


Figura 2.1: Arquitectura de un motor de flujos de trabajo propuesto por Rodriguez & Buyya [174].

permiten escalar los componentes del flujo en múltiples máquinas para acelerar su ejecución [120].

La paralelización de un flujo de trabajo se realiza en dos etapas: análisis y planificación. En la etapa de análisis se identifican las tareas, nodos y componentes que se pueden ejecutar de forma paralela. Mientras que, en la etapa de planificación, se elige la infraestructura donde se desplegarán estos componentes, así como su grado de paralelismo (número de trabajadores/hilos, patrón a utilizar y esquema de escalamiento) [120]. En este sentido, en la literatura se han identificado dos niveles de paralelismo [120]: *i*) paralelismo de grano grueso y *ii*) paralelismo de grano fino.

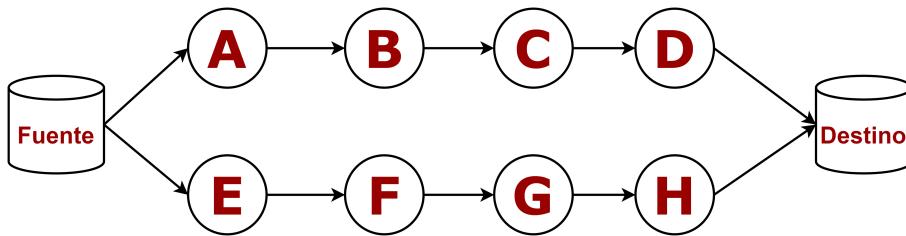


Figura 2.2: Representación conceptual de un flujo de trabajo compuesto por subflujos.

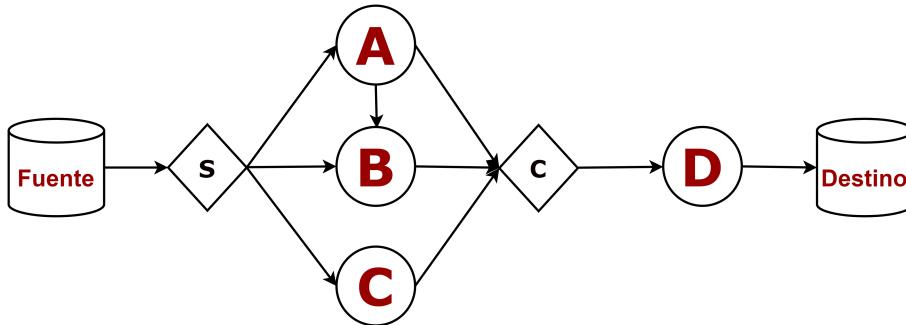


Figura 2.3: Representación conceptual de un flujo de trabajo con paralelismo de datos.

2.4.1.1. Paralelismo de grano grueso

El paralelismo de grano grueso se realiza a nivel de flujo de trabajo, mediante la identificación de subflujos de trabajo que pueden ser ejecutados de forma paralela. Por ejemplo, la Figura 2.2 muestra una representación conceptual de un flujo de trabajo compuesto de dos subflujos: el primero integrado por los nodos A-B-C-D, y el segundo por los nodos E-F-G-H. Estos subflujos de trabajo serán ejecutados paralelamente con este tipo de paralelismo.

2.4.1.2. Paralelismo de grano fino

El paralelismo de grano fino es realizado dentro un flujo de trabajo o un subflujo de trabajo. En este paralelismo, las etapas son paralelizadas utilizando técnicas como el paralelismo de datos, el paralelismo independiente, y el paralelismo de tuberías.

Paralelismo de datos. Se obtiene al tener varias tareas que realizan la misma actividad, cada una

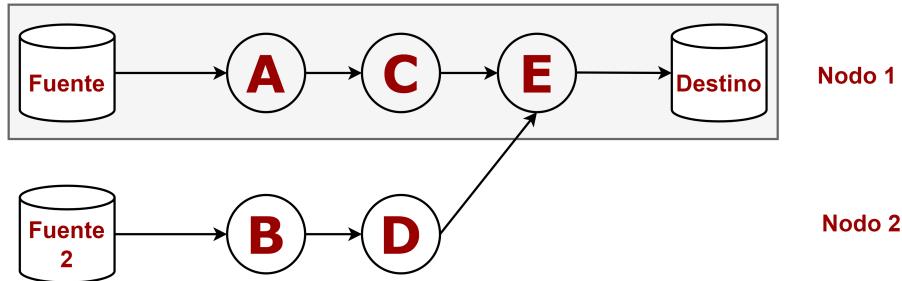


Figura 2.4: Representación conceptual de un flujo de trabajo con paralelismo de tareas.

en un segmento de datos diferente. La entrada de datos es dividida en diferentes segmentos y cada uno de estos, es procesado de forma independiente por una tarea en un procesador o nodo de cómputo diferente.

El paralelismo de datos puede ser estático, dinámico o adaptativo. Si el número de segmentos es definido durante el tiempo de configuración es estático, mientras que si el número de segmentos es definido durante tiempo de ejecución es dinámico. Finalmente, si el número de segmentos es definido durante tiempo de ejecución de acuerdo al ambiente de ejecución, quiere decir que el paralelismo es adaptativo [161].

La Figura 2.3 muestra una representación conceptual de esta técnica de paralelismo. Como se puede observar, un *segmentador* (*s*) es necesario para generar el segmento que será distribuido a los nodos. Además, se requiere un *consolidador* (*c*) para unir las salidas de los nodos. Estos componentes se encuentran representados por un rombo en la Figura 2.3.

Paralelismo de tareas. Diferentes actividades de un flujo de trabajo pueden ser ejecutadas paralelamente sobre varios nodos de cómputo. La ejecución de cualquiera de las etapas no depende de la salida de datos de otra. El paralelismo de tareas se logra al tener tareas de diferentes actividades independientes ejecutadas simultáneamente. La Figura 2.4 muestra la representación conceptual de este tipo de paralelismo donde los nodos A-C y los nodos B-D son ejecutados paralelamente en dos equipos de cómputo diferentes.

2.4.2 Calendarización de flujos de trabajo

La calendarización de los flujos de trabajo es un proceso realizado en los sistemas de gestión de flujos de trabajo, en el cual los nodos y las tareas de un flujo de trabajo son desplegados en los recursos computacionales (p. ej. máquinas virtuales en la nube) [33]. La meta de este proceso es obtener un plan de ejecución eficiente que minimice o maximice una o varias funciones objetivas, como minimizar los costos monetarios asociados a la ejecución del flujo de trabajo, maximizar el uso de recursos, minimizar la latencia entre componentes y maximizar la seguridad de los datos [174].

Los algoritmos de calendarización pueden ser clasificadas de acuerdo con los tipos de flujos de trabajo a calendarizar. En este sentido, se han identificado los siguientes tipos de flujos de trabajo [174]:

Flujo de trabajo simple. Es el modelo tradicional utilizado en el “grid computing” y en clústeres.

En este modelo, la ejecución de los flujos de trabajo es secuencial e independiente. Los requerimientos de calidad de servicio son diseñados por un solo usuario y un solo DAG.

Agrupamiento de flujos de trabajo. En este modelo los flujos de trabajo son agrupados para producir la salida deseada. Estos flujos de trabajo comparten una estructura similar, pero los datos a procesar son diferentes.

Múltiples flujos de trabajo. Los flujos de trabajo varían en su estructura y datos. Además, se desconoce la cantidad de flujos de trabajo que se desplegarán.

2.5 Tuberías de procesamiento de datos

Una tubería de datos está conformada por un conjunto de pasos que permiten el manejo y procesamiento de datos desde una o múltiples fuentes. En este sentido, la salida de un paso es la entrada del siguiente, y cada uno de estos pasos está a cargo de entregar los datos hasta que estos



Figura 2.5: Tubería tradicional de procesamiento de datos. Imagen extraída de <https://hazelcast.com/glossary/data-pipeline/>.

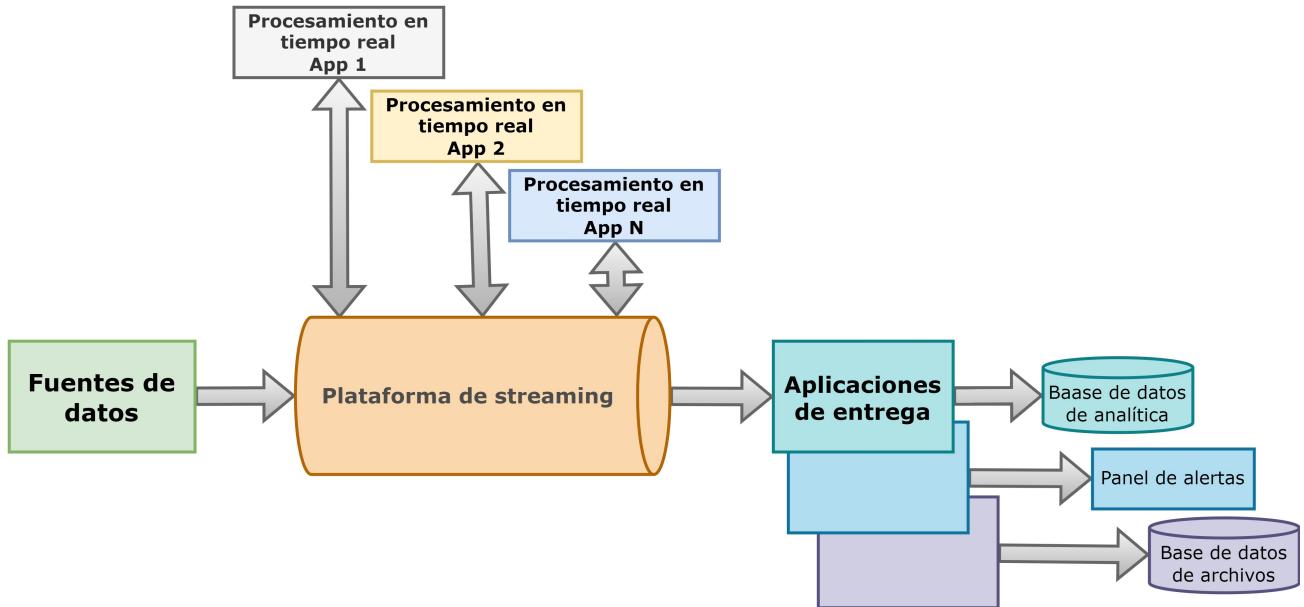


Figura 2.6: Tubería de datos para el procesamiento de datos en tiempo real. Imagen extraída de <https://hazelcast.com/glossary/data-pipeline/>.

lleguen a su destino [142].

El objetivo de una tubería de datos es crear un flujo de datos desde una o múltiples fuentes de datos hasta uno o múltiples destinos de datos (también llamados sumideros de datos). Por lo tanto, una tubería de datos se compone de tres elementos principales: fuentes, pasos de procesamiento y destinos. En la Figura 2.5 se muestra la representación conceptual de los componentes de una tubería de datos.

Las tuberías de datos pueden emplearse para resolver varios problemas de big data, por ejemplo, el manejo de grandes volúmenes de datos utilizando tuberías de datos que puedan escalar en el tiempo

para manejar la variabilidad de los datos. En este caso, a diferencia de un modelo ETL que solo puede implementarse para procesar datos por lotes (batch processing), las tuberías de datos permiten el diseño de estructuras para el procesamiento de datos en tiempo real (streaming) o bajo un esquema híbrido. En la Figura 2.6 se muestra la representación conceptual de una tubería de datos para el procesamiento en tiempo real, en donde las aplicaciones o pasos se ordenan en la forma de una tubería de procesamiento de múltiples fuentes.

2.6 Big data

En la literatura, el término *big data* ha tomado distintas definiciones para describir una variedad de conceptos y herramientas. El NIST define el *big data* como un conjunto de datos que no puede ser manejado de forma eficiente con las arquitecturas tradicionales [71]. Estos conjuntos de datos poseen distintas características, las cuales son conocidas como los “Vs” del *big data*:

- Volumen: el tamaño de los conjuntos de datos;
- Variedad: los datos son adquiridos desde diversas fuentes, repositorios y dominios;
- Velocidad: la velocidad del flujo a la que se producen los datos;
- Variabilidad: los cambios en los datos o en sus características.

En este contexto, el *big data* requiere una arquitectura escalable (p. ej. arquitecturas del cómputo en la nube) que permita el almacenamiento, manejo, procesamiento, y la visualización de los datos con esas características [71].

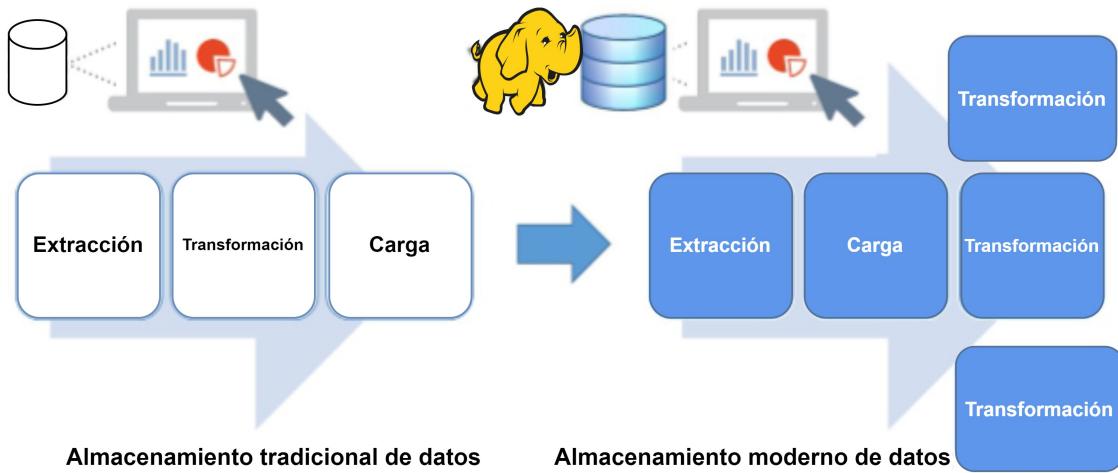


Figura 2.7: Comparación de los modelos ETL y ELTTT. Imagen extraída de [148].

2.6.1 Procesamiento por lotes

El procesamiento por lotes (*batch processing* en inglés) se refiere a la ejecución de aplicaciones o programas utilizados para procesar conjunto de tareas repetitivas sobre grandes volúmenes de datos [70]. Estos datos usualmente son estáticos, por ejemplo, videos de seguridad, bancos de datos o datos gubernamentales [40, 199].

Las soluciones de procesamiento por lotes están centradas en hacer factible el procesamiento de grandes volúmenes de datos mediante el escalamiento de las aplicaciones a través de múltiples nodos de procesamiento. En este sentido, estas soluciones extienden el modelo tradicional de extracción, transformación y carga (ETL, por su definición en inglés) escalando la operación de transformación, permitiendo el procesamiento de datos en paralelo [67]. Este modelo extendido es llamado ELTTT y la etapa de carga es realizada antes de la etapa de transformación para preservar los datos originales en la base de datos accesible por todos los nodos de transformación distribuidos (ver Figura 2.7 para una comparación conceptual del modelo tradicional ETL y el modelo ELTTT) [59].

Ejemplos de herramientas de procesamiento por lotes incluyen Hadoop, el cual es un marco de trabajo

	Procesamiento por lotes	Procesamiento de stream
Tipo de datos	Consultas o procesamiento sobre todos o la mayoría de los datos en un conjunto de datos.	Consultas o procesamientos sobre los datos dentro de una ventana de tiempo móvil o solo en el registro de datos más reciente.
Tamaño de los datos	Grandes lotes de datos.	Registros individuales o micro lotes que consisten en unos pocos registros.
Rendimiento	Latencias en un rango de minutos a horas.	La latencia se encuentra en el orden de los segundos a los milisegundos.
Análisis	Analítica compleja	Respuesta de función simple, agregados y métricas continuas.

Figura 2.8: Comparación del procesamiento en lotes y en tiempo real. Imagen extraída de <https://aws.amazon.com/blogs/bigdata/implement-serverless-log-analytics-using-amazon-kinesis-analytics/>.

de código abierto para el procesamiento de grandes volúmenes de datos. Hadoop incluye el modelo de programación MapReduce [117], en el cual los datos son procesados en dos etapas:

Map: donde la entrada de los datos es cargada y transformada de forma paralela.

Reduce: la fase de reducción, donde los registros asociados serán procesados por la misma entidad.

2.6.2 Procesamiento en tiempo real

El procesamiento en tiempo real se centra en el manejo y procesamiento de datos producidos en tiempo real y la velocidad en la cual estos datos son procesados. La meta es reducir la latencia para proveer una retroalimentación a los usuarios lo más pronto posible [121]. Los datos producidos en tiempo real son aquellos datos generados de forma continua por diferentes fuentes heterogéneas (p.

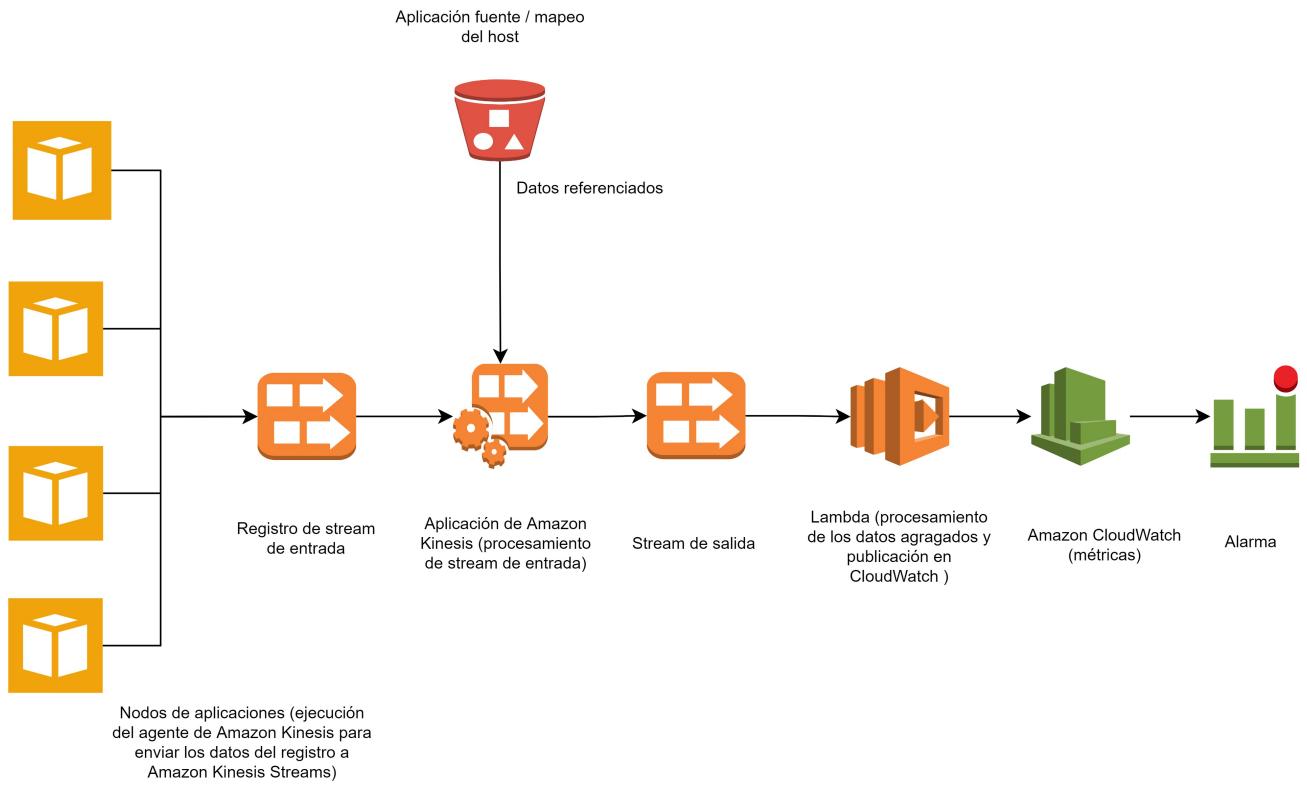


Figura 2.9: Representación conceptual de Amazon Kinesis. Imagen extraída de <https://aws.amazon.com/blogs/big-data/implementserverless-log-analytics-using-amazon-kinesis-analytics/>.

ej. diferentes dispositivos de IoT). Los datos generados por estas fuentes son de tamaño “pequeño” (se encuentran en el orden de los kilobytes), y necesitan ser procesados secuencialmente, comúnmente sobre ventanas de tiempo [176]. Ejemplo de datos de tiempo real son aquellos datos producidos por los vehículos, equipo industrial, dispositivos de IoT, monitores de compañías eléctricas, y videos y juegos en línea [176].

La Figura 2.8 describe las principales diferencias entre el procesamiento por lotes centrado en el procesamiento de grandes volúmenes de datos y el procesamiento en tiempo real.

Diferentes herramientas han sido propuestas para apoyar este tipo de procesamiento de datos, las cuales incluyen Amazon Kinesis [211] y Spark Streaming [166]. La Figura 2.9 muestra el diseño conceptual de Amazon Kinesis, el cual recibe datos desde distintas fuentes, los cuales son enviados a

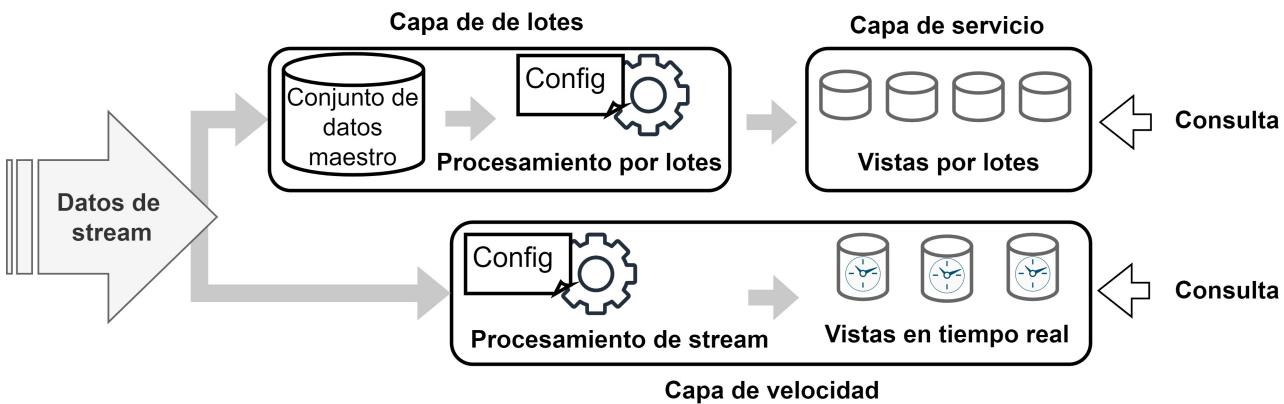


Figura 2.10: Arquitectura Lambda. Imagen extraída de Hausenblas, M., & Bijnens, N. (2015) [90].

Amazon Lambda para agregar los resultados, y hacer que los datos sean disponibles para los usuarios finales [211].

2.6.3 Procesamiento híbrido

El procesamiento híbrido es una combinación del procesamiento por lotes y el procesamiento en tiempo real. En este enfoque el objetivo es el procesamiento tanto de los lotes como de los datos producidos en tiempo real, combinando de esta forma los resultados de ambas técnicas de procesamiento con una baja latencia durante la entrega de retroalimentación para los usuarios finales [90]. Para esto, la entrada de datos se encuentra distribuida en dos capas llamadas capa de lotes y capa de velocidad [90]. La Figura 2.10 presenta estas capas organizadas como una arquitectura conocida como arquitectura Lambda.

En la capa de lotes existen aplicaciones tradicionales de procesamiento ETL o ELTTT. En esta capa los datos son cargados a un conjunto de datos maestro, el cual incluye los registros de todos los datos entrantes históricos. La meta de esta capa es obtener información del procesamiento del conjunto de datos completo. Los resultados de esta capa son entregados a la capa de servicio, la cual incluye un conjunto de vistas por lotes para que los resultados estén disponibles para las aplicaciones externas

y los usuarios finales [90]. Por otro lado, en la capa de velocidad, todas las aplicaciones realizan el procesamiento de los datos en tiempo real para entregar una retroalimentación rápida a los usuarios finales a través de vistas en tiempo real [90].

Algunas herramientas disponibles en la literatura que permiten el procesamiento híbrido de los datos son Apache Flink [37] y la arquitectura de Amazon Lambda [90].

2.7 Red de distribución de contenidos

Una red de distribución de contenidos (CDN, por sus siglas en inglés) es un sistema de almacenamiento distribuido compuesto por un conjunto de equipos de almacenamientos distribuidos geográficamente. El objetivo principal de una CDN es reducir la latencia observada para acceder a contenidos a través de una red (p. ej. Internet), utilizando servidores en el borde desplegados en centros de datos lo más cercano posible a los usuarios finales. [34]. En una CDN tradicional, los contenidos son replicados a través de diferentes servidores, y los usuarios acceden a los contenidos utilizando una *URI* que redirecciona al usuario al servidor más cercano, con el objetivo de reducir el tiempo requerido para acceder a los contenidos. Para reducir este tiempo, las CDNs implementan técnicas de *caching*, balanceo de carga y compresión de datos [34, 160].

Una CDN tiene los siguientes objetivos basados en requerimientos de calidad de servicio (QoS, por su definición en inglés) [160]:

Escalabilidad. Una CDN debe tener la capacidad para crecer con el objetivo de poder atender un número mayor de solicitudes de los clientes, sin que los usuarios finales lo noten.

Seguridad. Una CDN debe cumplimentar requerimientos para garantizar los datos a manejar y la infraestructura donde está desplegada, para evitar riesgos de fuga de datos, ataques de denegación de servicios, virus y software malicioso.

Confiabilidad. Los datos en una CDN deben de ser accesibles aún en casos de fallas en la infraestructura de almacenamiento. En este sentido, comúnmente una CDN utiliza múltiples infraestructuras de almacenamiento distribuidas geográficamente, e implementa técnicas de confiabilidad basadas en codificación y replicación de datos (p. ej. el algoritmo de dispersión de información [167]).

2.8 Infraestructuras para el manejo de servicios

Una estructura de procesamiento puede ser desplegada en una sola infraestructura o de forma distribuida en múltiples infraestructuras. Infraestructuras tradicionales para el manejo de estructuras de procesamiento como flujos de trabajo, incluyen clústeres o equipos alto rendimiento, en los cuales se utilizan plataformas como HTCondor [69] o Slurm para el manejo y distribución de tareas en la infraestructura disponible. Por ejemplo, calendarizando las tareas para su ejecución en los equipos que conforman el clúster o manejando el uso de los núcleos físicos de estos equipos.

Las estructuras de procesamiento y sus componentes comúnmente se despliegan en diferentes infraestructuras dependiendo de los requerimientos de las organizaciones. Ejemplos de infraestructuras comúnmente utilizados incluyen clústeres de alto desempeño, sistemas distribuidos descentralizados como el *grid computing*, la nube o el cómputo borde-niebla-nube (edge-fog-cloud en inglés), los cuales han sido utilizados para el despliegue a gran escala de las etapas contempladas en una estructura de procesamiento.

2.8.1 Clústeres de alto rendimiento

Un clúster de alto rendimiento (HPC, por sus siglas en inglés) está compuesto por un conjunto de equipos de computación que se encuentran organizados e interconectados para funcionar como un

solo sistema. Estos equipos comúnmente tienen múltiples procesos que permiten el despliegue de aplicaciones paralelas para resolver problemas que requieren de altas prestaciones de cómputo. Por ejemplo, experimentos, el despliegue y ejecución de flujos de trabajo utilizados para experimentos en áreas como la astronomía o la genética, que consideran la ejecución de miles de instancias.

Los equipos en un clúster de alto rendimiento se encuentran interconectados a través de la red y comparten tanto recursos de hardware como de software (p. ej. el sistema de archivos). Estos clústeres son construidos utilizando gestores de la infraestructura tales como HTCondor [69], Slurm [100] o gestores de contenedores como Docker Swarm [193] o Kubernetes [124].

2.8.2 Grid computing

En grid computing, un conjunto de equipos heterogéneos pertenecientes a diferentes organizaciones son interconectados mediante la red (comúnmente Internet) creando una *supercomputadora virtual*. En este enfoque, no solo se interconectan equipos de procesamiento (como en clúster), sino que además permite la conexión de equipos de almacenamiento, bases de datos, red, procesadores y aplicaciones, permitiendo el despliegue de aplicaciones de alto rendimiento para procesar grandes volúmenes de datos. Sin embargo, la interconexión de equipos de diferentes organizaciones puede ser compleja y en algunos casos complicada por las diferentes políticas y normativas impuestas por cada organización. Otra desventaja de este tipo de infraestructura es que las aplicaciones deben de ser modificadas para adaptarse y utilizar eficientemente los recursos disponibles.

2.8.3 Cómputo en la nube

El NIST define la nube como “un modelo para habilitar el acceso de red ubicuo, conveniente y bajo demanda a un conjunto compartido de recursos (p. ej. redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser aprovisionados y liberados rápidamente con un mínimo

esfuerzo de gestión o interacción con el proveedor de servicios” [134]. De forma similar, Buyya et al. definen la nube como un tipo de sistema distribuido y paralelo que está conformado por un conjunto de computadoras virtualizadas e interconectadas, que para el usuario son presentadas por un proveedor como un conjunto único de recursos coherentes y unificados [35]. En este contexto, el cómputo en la nube puede ser dividido en tres modelos principales:

- Infraestructura como servicio (IaaS): en este modelo, los usuarios pueden acceder a distintos recursos de almacenamiento, cómputo y redes virtualizadas bajo demanda. Los usuarios tienen acceso a servicios de front-end desde los cuales ellos pueden crear recursos virtualizados bajo demanda, los cuales son proporcionados por un proveedor de servicios en la nube (p. ej. Amazon EC2 o Google Cloud) [183]. En las IaaS, los usuarios manejan la implementación de aplicaciones (incluyendo la instalación de dependencias), así como la entrega de los datos a las instancias de la nube. Además, los usuarios tienen que gestionar la ejecución de las aplicaciones a través de un middleware que ellos tienen que desarrollar.
- Plataforma como servicio (PaaS): en este modelo, un conjunto de aplicaciones basadas en la nube (p. ej. sistemas para la gestión de bases de datos, herramientas de desarrollo, y la web) están disponibles para que los usuarios creen software en la nube. En los PaaS, los usuarios solamente se preocupan por cómo serán desplegadas sus aplicaciones y como será realizada la carga de sus datos en la nube [116].
- Software como servicio (SaaS): en este modelo, una aplicación es entregada a los usuarios como un servicio (comúnmente es accesible a través de un navegador web). En este modelo, los usuarios no tienen ningún trato con la gestión de la infraestructura y el despliegue de las aplicaciones. Ejemplo de soluciones SaaS son Dropbox, Google Workspace y Microsoft Teams [65].

Dependiendo del tipo de acceso, la nube se puede clasificar en tres tipos:

Nube pública. Los recursos virtualizados son desplegados sobre infraestructura compartida, la cual es provista para uso abierto del público en general. Esta infraestructura comúnmente es operada y manejada por una organización pública o privada, la cual es llamada proveedor de servicios en la nube. Ejemplos de estos proveedores son Amazon EC2, Google Cloud y Microsoft Azure. [134, 189].

Nube privada. Los recursos virtualizados son desplegados en infraestructura que es de uso exclusivo de una sola organización que cuenta con múltiples consumidores (p. ej. unidades de negocio). Esta organización está a cargo de mantener en operación la infraestructura, la cual pertenece a ellos [74, 134].

Nube híbrida. Diferentes infraestructuras públicas y privadas son agrupadas como una sola nube utilizando tecnología estandarizada y/o propietaria, la cual permite la portabilidad de los datos y aplicaciones [134, 216].

Para poder procesar los grandes volúmenes de datos que producen sus usuarios, las organizaciones han empezado a migrar sus servicios y datos a la nube [101, 168, 187]. La nube permite que las organizaciones tengan acceso a recursos y servicios de forma rápida en cualquier lugar y momento, cuando mejor convenga a las necesidades de la organización [14]. Además, su modelo de pago por uso, conocido en inglés como pay-as-you-go, es muy atractivo para las organizaciones, dado que les permite reducir costos en adquisición y manejo de infraestructura [98, 218].

La nube ha surgido como una evolución de grid computing, en el cual las organizaciones pueden acceder a una piscina de recursos virtualizados (p. ej. almacenamiento, procesamiento o red), los cuales pueden ser adquiridos bajo demanda a través de la red. Este enfoque se ha vuelto popular dado que permite a las organizaciones escalar sus recursos disponibles de forma sencilla y en algunas ocasiones de forma automática, pagando solo por aquellos recursos utilizados en el caso de la nube pública (pay-as-you-go) [98, 218]. Por lo tanto, la nube permite a las organizaciones manejar el ciclo de vida de sus productos digitales de forma distribuida y creando servicios a los cuales los usuarios

finales pueden acceder a través de la red.

Sin embargo, cuando las organizaciones utilizan solo recursos virtuales de un proveedor en la nube, estos pueden generar una dependencia con este proveedor (vendor lock-in), lo cual puede generar problemas durante la migración de servicios y datos entre infraestructuras de diferentes proveedores, debido a que cada proveedor utiliza tecnología propietaria para el manejo de las máquinas virtuales donde se despliegan las aplicaciones y se almacenan los datos, lo cual no hace factible que una imagen de una máquina virtual pueda ser descargada y desplegada en los recursos de otro proveedor en la nube. Además, con el incremento del número de dispositivos conectados y produciendo información (p. ej. dispositivos de Internet de las Cosas, sensores y dispositivos móviles) así como la dispersión geográfica de los usuarios que consumen los productos digitales, han causado que el uso de un solo proveedor en la nube en una región en específico no sea factible.

Debido a lo anterior, se han propuesto diferentes enfoques para distribuir el procesamiento y manejo de los productos digitales entre diferentes proveedores en la nube y entornos de infraestructura. Ejemplos de estos enfoques son las soluciones nube a nube (C2C, por sus siglas en inglés) que implica hacer respaldo de los servicios y datos utilizando diferentes proveedores en la nube o soluciones jerárquicas como el cómputo *borde-niebla-nube* en el que se hace uso de infraestructuras intermedias entre la nube y el usuario final para reducir el tiempo de respuesta para obtener resultados cuando se ejecuta una estructura de procesamiento.

2.8.4 Cómputo borde-niebla-nube

En años recientes, el número de dispositivos conectados a internet que producen datos ha aumentado, esto como consecuencia de nuevos paradigmas emergentes como los es el Internet de las Cosas (IoT) [204]. En este sentido, una colección centralizada de recursos en la nube para el manejo de grandes volúmenes de datos se vuelve inviable, lo anterior se debe a los costos que representaría manejar esta

gran cantidad de recursos.

Para hacer frente a este problema, el manejo, almacenamiento y procesamiento de los datos comúnmente es realizado a través de un enfoque jerárquico. El IDC organiza esta jerarquía en tres capas conocidas como extremo, borde y núcleo (core) [170]. El núcleo hace referencia a todos aquellos centros de cómputo, incluyendo las nubes públicas, privadas e híbridas. En el núcleo, los datos son procesados y almacenados para obtener información que ayude a los usuarios en los procedimientos de toma de decisiones. Por otro lado, el borde se refiere a todos aquellos servidores y centro de datos pequeños que son manejados por las organizaciones. En esta capa, los datos son procesados para prepararlos y ser almacenados en la nube u obtener resultados rápidamente. Finalmente, la capa de extremo incluye todos aquellos dispositivos que producen datos en el borde de la red. Entre estos dispositivos se incluyen las computadoras, los celulares, los dispositivos de IoT, y los sensores.

En la literatura, esta organización jerárquica es conocida como cómputo en el *borde-niebla-nube*, donde los paradigmas de borde y niebla son utilizados para mitigar la saturación de los recursos en la nube y reducir el tiempo de respuesta de los sistemas [136]. En el cómputo en el borde, una colección de dispositivos informáticos de uso general (p. ej. celulares o computadores personales) son interconectados con sensores [186] para reducir la latencia producida al enviar los datos directamente a la nube o niebla [136, 158, 224].

El cómputo en la niebla [223], incluyendo el almacenamiento en la niebla [143], se refiere a todos aquellos servidores y centros de datos pequeños que se encuentran localizados cerca del borde de la red. Los dispositivos de la niebla reciben los datos desde los nodos del borde [31, 197]. Los datos son almacenados y procesados para generar información, que en los nodos de la niebla, ayude en los procedimientos de toma de decisiones. Finalmente, los datos y los resultados se envían a la nube para su almacenamiento y posterior procesamiento [54, 143].

En este sentido, desde el punto de vista del usuario final, una estructura de procesamiento desplegada en el borde-niebla-nube es vista como una caja negra, en la que sus componentes pueden estar

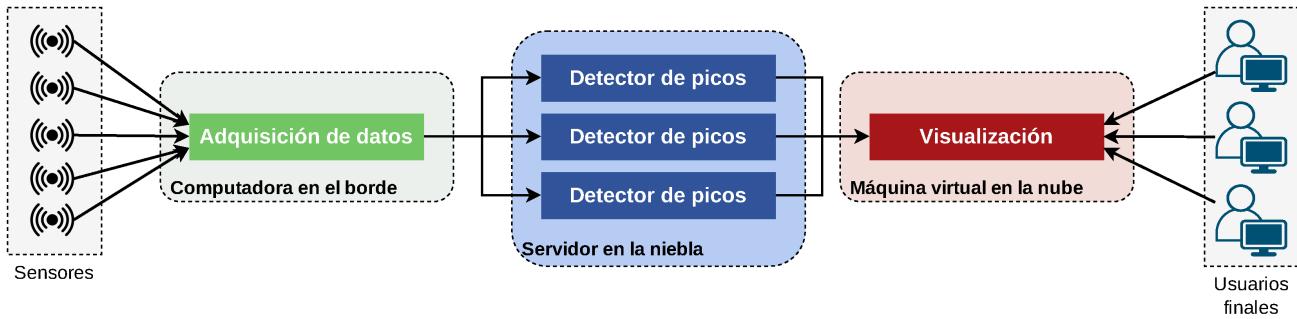


Figura 2.11: Ejemplo de una estructura de procesamiento desplegado en el borde-niebla-nube.

transparentemente distribuidos en diferentes infraestructuras. Esta caja negra recibe un conjunto de productos digitales de entrada y le entrega un conjunto de resultados (comúnmente utilizando herramientas de visualización de datos). En la Figura 2.11 se observa un ejemplo de una estructura de procesamiento para el manejo de señales de electrocardiograma capturadas por un conjunto de sensores. Como se puede observar, el procesamiento de estas señales es visto como una caja negra por el usuario final. La estructura de procesamiento implementada incluye tres etapas para la adquisición de las señales en el borde, su procesamiento utilizando un detector de picos en la niebla y su visualización en la nube.

2.8.5 Mecanismos de despliegue de aplicaciones en la infraestructura

Para materializar una aplicación en una infraestructura, comúnmente se requiere de un *contenido* que es ejecutado en un *contexto*. El contenido incluye el código de la aplicación (creado en lenguajes de programación como C/C++, Python o Java) así como sus parámetros de ejecución. Por su parte, el contexto es definido por las características de la plataforma y la infraestructura donde la aplicación es desplegada. Dicho contexto, suele estar definido en la forma de archivos de configuración que incluyen las características de la plataforma (p. ej. sistema operativo) y la infraestructura (p. ej. almacenamiento, memoria RAM o número de núcleos).

En este sentido, este despliegue de aplicaciones se puede realizar de forma manual o automática [88].

En el primer caso, los desarrolladores deben de instalar y configurar las aplicaciones en la infraestructura donde se desplegará la estructura de procesamiento. Lo anterior incluye la instalación y configuración de dependencias (p. ej. interprete/compilador) y variables de entorno. Esto puede generar una dependencia con la plataforma donde se desplieguen las aplicaciones, debido a que este proceso de instalación de aplicaciones puede ser costoso en tiempo y en algunas ocasiones costoso en términos monetarios (p. ej. cuando se usa la nube pública), lo cual ocasiona que las organizaciones no puedan desplegar sus aplicaciones en otras plataformas.

Dado lo anterior, se han desarrollado soluciones que permiten el despliegue automático de aplicaciones sin requerir la instalación y configuración de las estas y sus dependencias. La tecnología de contenedores virtuales ha vuelto factible la construcción de paquetes de software, en los cuales se instalan la aplicación y sus dependencias. Este proceso de construcción solo es realizado una vez, y genera un paquete de software listo para desplegarse en múltiples infraestructuras y plataformas [226].

2.9 Contenedores virtuales

Un contenedor virtual es una unidad de software que empaqueta el código y dependencias (p. ej. bibliotecas del sistema, paquetes de terceros, o archivos de configuración) de una aplicación. Estos contenedores tienen las características de que son ligeros, en términos de espacio de almacenamiento y en comparación con máquinas virtuales tradicionales, así como que se encuentran aislados del resto de aplicaciones en el sistema operativo anfitrión [28].

2.9.1 Docker

Docker es el gestor de contenedores virtuales más popular actualmente. Permite automatizar la construcción, despliegue y manejo de contenedores virtuales, los cuales encapsulan todo lo necesario

```

1. FROM python:3.7.6-buster ← Imagen base para construir el contenedor.
2.
3. WORKDIR /installation
4. ADD requirements.txt .
5. RUN pip install -r requirements.txt ← Instalación de requerimientos.
6.
7. WORKDIR /app ← Directorio de trabajo del contenedor.
8. ADD /code .
9.
10. ADD /API /API ← Copiado de archivos desde el equipo
    anfitrión al contenedor.
11.
12. EXPOSE 5000
13. ENTRYPOINT ["python3", "/API/main.py"] ← Comando de ejecución al iniciar el
    contenedor

```

Figura 2.12: Ejemplo de un archivo Dockerfile.

para que una aplicación pueda ser desplegada exitosamente en una infraestructura.

Docker utiliza directamente el kernel de Linux para construir contenedores virtuales, por lo cual no es necesario virtualizar un sistema operativo para que estos puedan ser desplegados, como es el caso de las máquinas virtuales tradicionales. Lo anterior reduce el tamaño de los contenedores virtuales en comparación de las máquinas virtuales, lo cual permite que estos puedan ser portados entre diferentes infraestructuras.

Desde el punto de vista del sistema operativo anfitrión, un contenedor virtual es manejado como un proceso independiente, permitiendo el despliegue de múltiples contenedores en la misma infraestructura.

En Docker los contenedores son construidos a partir de imágenes, las cuales básicamente son paquetes de software que contienen todo lo necesario para permitir el despliegue de un contenedor. En este caso, Docker permite la creación de imágenes de contenedor escribiendo archivos de configuración llamados *Dockerfiles*, los cuales contienen una serie de comandos para generar una imagen de contenedor. En la Figura 2.12 se muestra un ejemplo de un Dockerfile para la encapsulación de una aplicación escrita en el lenguaje de programación Python 3.7.6. Este archivo es interpretado por Docker, generando

una imagen de contenedor que al ser instanciada generará un contenedor virtual desde donde se podrá ejecutar la aplicación contenida.

Algunas de las ventajas del uso de contenedores Docker, son las siguientes:

- Los contenedores Docker permiten manejar modularmente los componentes de un sistema, facilitando su manejo y despliegue sin afectar el rendimiento del resto de componentes.
- Docker facilita a los desarrolladores tener un control sobre las diferentes versiones de una aplicación. Lo anterior es gracias a que las imágenes Docker están basadas en capas que se van agregando a una imagen base. Esto también permite, que los desarrolladores y usuarios puedan volver a una versión anterior de la imagen.
- Los contenedores virtuales de Docker pueden ser fácilmente desplegados en una infraestructura, reduciendo los costos y esfuerzos asociados con configurar una aplicación y sus dependencias.
- Los contenedores virtuales tienen un alcance limitado, que permite instalar y configurar múltiples aplicaciones en diferentes aplicaciones sin afectar el funcionamiento del resto de contenedores y de las aplicaciones instaladas directamente en el sistema operativo anfitrión.

Cabe resaltar, que los datos manejados al interior de un contenedor son efímeros, por lo que al eliminar este contenedor el acceso a los datos se perderá. En este sentido, el desarrollador es responsable de crear volúmenes de datos que permitan compartir datos entre el contenedor y el sistema anfitrión.

2.9.1.1. *Docker Compose*

Docker Compose es una extensión de Docker que permite el manejo de sistemas construidos con múltiples contenedores. En Docker Compose los contenedores son manejados como servicios que son declarados en un archivo YML. En este sentido, la creación de un sistema con Docker Compose se realiza en tres pasos:

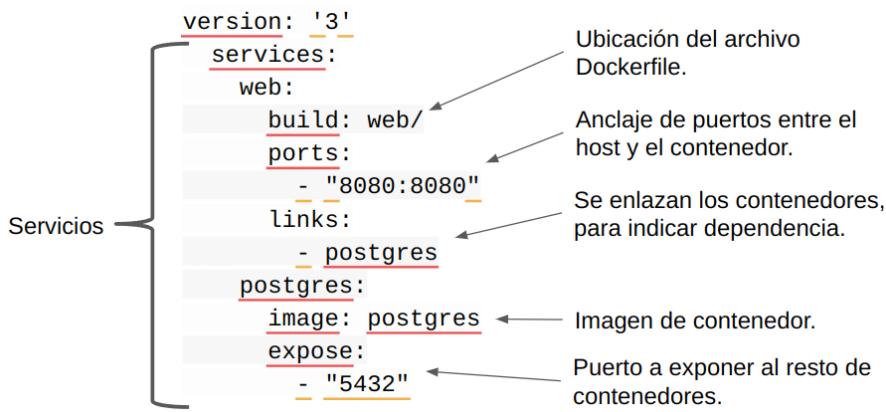


Figura 2.13: Ejemplo de un archivo YML utilizando en Docker Compose.

1. Diseño de las imágenes de contenedor de cada servicio que conforman el sistema. En este caso, los desarrolladores deben de escribir los archivos Dockerfile que permitan abstraer un conjunto de aplicaciones como imágenes de contenedor.
2. Definición de un archivo YML que contenga la declaración de los servicios, así como de sus configuraciones (p. ej. volúmenes, variables de entorno y puertos). En la Figura 2.13 se muestra un ejemplo de un archivo YML en el cual se declaran dos servicios: un servicio web que contiene una aplicación que se ejecuta sobre un servidor Apache, y una base de datos de PostgreSQL.
3. Despliegue del sistema mediante la instancia de los servicios declarados en el archivo YML.

Docker Compose tiene las siguientes características principales:

- Aislar múltiples servicios y sistemas en un solo anfitrión, sin afectar el funcionamiento del resto.
- Manejo de volúmenes para la preservación de datos, incluso cuando una nueva versión de los contenedores es desplegada.
- Permite el manejo de las versiones de un contenedor, recreando solo aquellos servicios que han sido modificados sin afectar el funcionamiento del resto de servicios del sistema.

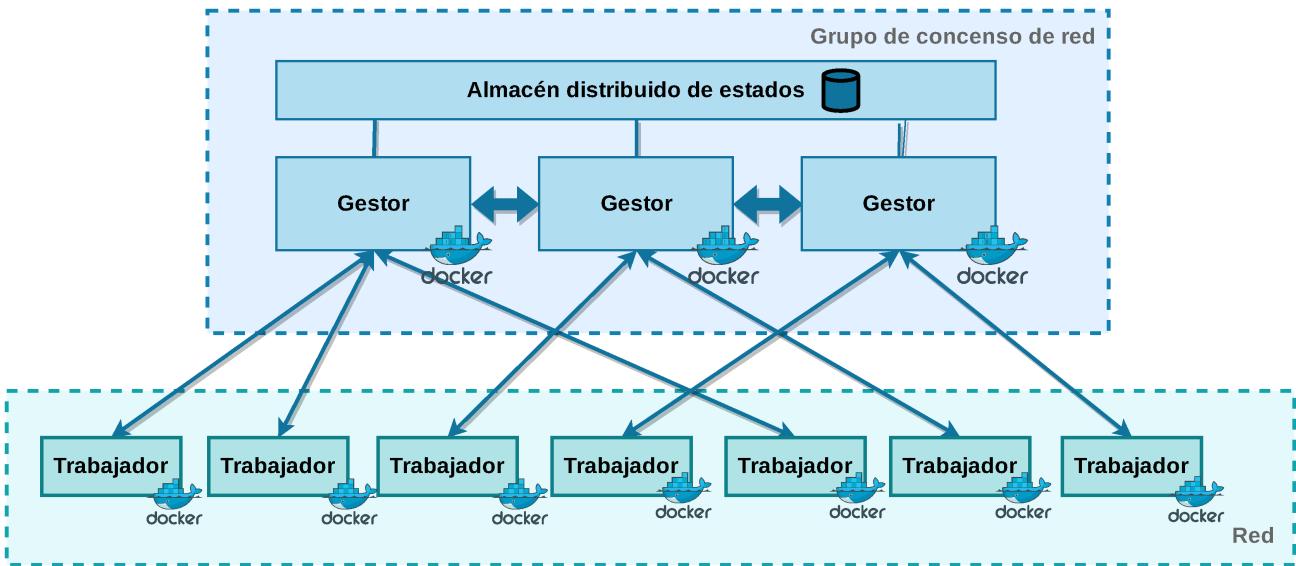


Figura 2.14: Clúster de Docker Swarm. Imagen extraída de <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>.

2.9.1.2. Docker Swarm

Docker Swarm es una herramienta que permite la orquestación de contenedores Docker en un clúster de computadoras. Similar a Docker Compose, Swarm permite el despliegue y manejo de sistemas conformados por múltiples contenedores. Estos sistemas se declaran utilizando archivos YML que incluyen la configuración de cada sistema considerado. Un clúster de Docker Swarm está conformado por múltiples equipos físicos o máquinas virtuales que pueden tomar el rol de gestor, líder o trabajador.

El *gestor* está a cargo de asignar tareas a los nodos en el clúster. Un clúster de Swarm puede incluir múltiples nodos gestores, entre los cuales se elige un *líder* utilizando el algoritmo de consenso Raft [149]. El nodo líder se encarga de realizar las tareas de manejo y orquestación en el clúster. En este sentido, este nodo está a cargo de distribuir las tareas de despliegue de los contenedores en los nodos trabajadores. Por otro lado, los nodos trabajadores están a cargo de recibir y materializar las tareas de manejo de contenedores asignadas por los nodos gestores.

En la Figura 2.14 se muestra la representación conceptual de un clúster de Docker Swarm compuesto

por tres nodos gestores y siete trabajadores. Todos los nodos en un clúster implementan un balanceador de carga para distribuir las solicitudes de los usuarios entre los contenedores y nodos en el clúster.

2.9.2 Kubernetes

Kubernetes es una plataforma de código abierto que permite la automatización de las tareas de orquestación, calendarización, despliegue y escalamiento de contenedores virtuales en un clúster [124]. Docker y Kubernetes pueden ser utilizados en conjunto. Mientras que Docker permite encapsular una aplicación y todas sus dependencias en un contenedor virtual, el cual puede ser desplegado en múltiples infraestructuras, Kubernetes permite gestionar estos contenedores en tiempo de ejecución. Desde el punto de vista de la industria, Docker es el estándar en la creación y distribución de aplicaciones en contenedores, mientras que Kubernetes se ha convertido en el estándar para el manejo de los contenedores durante su ciclo de vida, incluyendo servicios para:

- Desplegar contenedores virtuales en un equipo en específico y mantenerlo en ejecución.
- Manejo de las diferentes versiones de una aplicación durante su ciclo de vida: en desarrollo, pruebas o producción.
- Descubrimiento de servicios mediante la exposición de los contenedores a una red como Internet o a otros contenedores en una red interna.
- Manejo del almacenamiento de los contenedores, asignándoles un espacio de almacenamiento local o en la nube.
- Distribución de la carga de trabajo basado, por ejemplo, en la utilización del CPU.
- Manejo de cargas de trabajo entrantes escalando los contenedores con alta demanda.
- Recuperación de los contenedores virtuales en caso de que estos fallen.

Un clúster de Kubernetes este conformado por un conjunto de *nodos*, que pueden tomar el rol de maestro o trabajador. El *nodo maestro* se encarga de la calendarización de contenedores para su despliegue automático en los trabajadores, dependiendo de las capacidades de cómputo de estos trabajadores. Mientras que los *nodos trabajadores* se encargan de desplegar, ejecutar y manejar los contenedores virtuales. En Kubernetes, los contenedores se pueden organizar en *pods*, los cuales básicamente son grupos de contenedores que se despliegan en el mismo equipo y que comparten una red. En el caso de que un contenedor en pod sea altamente demandado, Kubernetes creará réplicas de ese pod en otros nodos del clúster.

2.9.3 Singularity

Singularity es un gestor de contenedores virtuales comúnmente utilizado para construir flujos de trabajo científicos que puedan ser reproducidos en múltiples sistemas. En este sentido, los contenedores virtuales de Singularity son muy parecidos a los Docker. La principal diferencia es que los contenedores en Singularity no requieren de privilegios de administrador para poder ser ejecutados, lo que agiliza su despliegue. Este gestor de contenedores es ejecutado como un proceso más del usuario, por lo que los contenedores desplegados con este gestor automáticamente tendrán acceso al sistema de archivos del usuario. En este sentido, un contenedor tiene acceso a su propio sistema de archivos (el cual por defecto es de solo lectura) y al sistema de archivos del usuario.

2.10 Cómputo sin servidor

Jonas et al. [105] describen que un *servicio sin servidor* (serverless) es aquel que tiene la capacidad de escalar automáticamente en la nube sin necesidad de que un aprovisionamiento explícito de los recursos. En este sentido, el desarrollo de servicios sin servidor se hace a alto nivel, donde el desarrollador debe de entregar una función y especificar el evento que lanzará la ejecución del servicio

(p. ej. la carga de un nuevo registro en la base de datos). La plataforma se hará cargo del manejo y ejecución del servicio, realizando la selección de la instancia (máquina virtual) para ejecutarlo, su escalamiento, despliegue, manejo de tolerancia a fallos, monitoreo, registro de eventos y manejo de la seguridad [17, 105].

Los servicios sin servidor surgieron como una manera de delegar el manejo del entorno de ejecución de las aplicaciones al proveedor de la nube, con el objetivo de reducir los tiempos que los desarrolladores invertían en configurar las máquinas virtuales donde estos ejecutarían su código [17, 105]. En este sentido, los servicios sin servidor buscan hacer frente los siguientes problemas que comúnmente tienen los desarrolladores cuando trabajan con la nube [105]:

- Redundancia y distribución geográfica de los servicios para evitar tener un único punto de falla.
- Balanceo de carga de las solicitudes para manejar eficientemente los recursos disponibles.
- Autoescalamiento de los servicios con base en cambios de la carga de trabajo entrante.
- Monitoreo y tolerancia a fallos de los servicios.
- Registro de eventos en una bitácora.
- Actualización y manejo de la seguridad del sistema.
- Migración de servicios a medida que nuevas instancias (máquinas virtuales) se encuentran disponibles.

El cómputo sin servidor es una variación del modelo de PaaS. En la que los desarrolladores y usuarios no tienen control sobre la configuración y administración de los recursos virtualizados donde se despliegan sus servicios, los cuales comparten la infraestructura con otros servicios. En la Figura 2.15 se puede observar el grado de control que tiene un desarrollador cuando utiliza cómputo sin servidor. Como se puede observar, a diferencia de IaaS donde el desarrollador no tiene control sobre los servicios ni la infraestructura, en el cómputo sin servidor los desarrolladores pueden crear

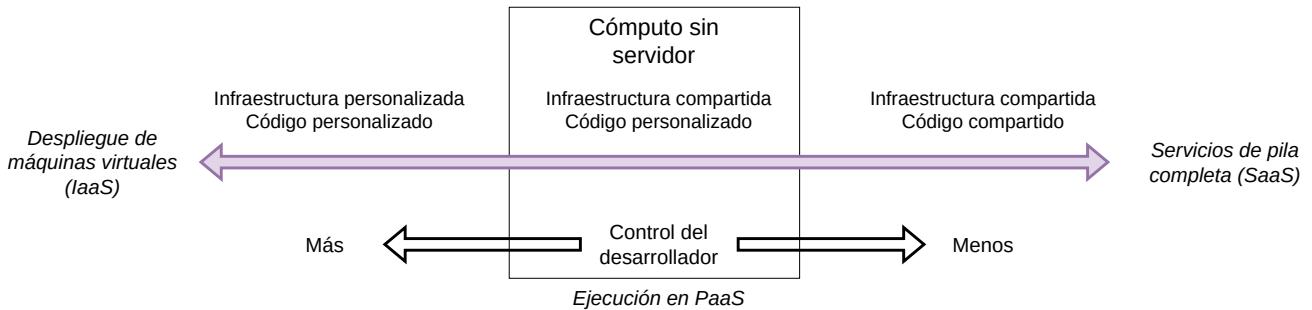


Figura 2.15: Comparación del control del desarrollador sobre el código de las aplicaciones y la infraestructura en cómputo sin servidor, IaaS y SaaS. Imagen extraída de [17].

sus propias aplicaciones y servicios que son desplegadas en una infraestructura compartida [17]. En el lado contrario, en IaaS el desarrollador tiene a su disposición infraestructura virtualizada personalizada donde puede desplegar sus aplicaciones. Sin embargo, en este caso el desarrollador tiene que configurar la infraestructura virtualizada para garantizar el funcionamiento de su aplicación. Este último enfoque es conocido como cómputo *serverful*, en el cual los desarrolladores tienen que administrar sus aplicaciones a bajo nivel. Las principales diferencias entre el cómputo sin servidor con serverful son las siguientes [105]:

- En cómputo sin servidor los recursos de almacenamiento y procesamiento se encuentran desacoplados, por lo que pueden escalar y aprovisionados de forma independiente.
- La asignación de recursos es realizada automáticamente por el proveedor en la nube cuando una aplicación es ejecutada con base a un evento, en lugar de solicitar recursos a priori.
- En la nube pública, el pago de los recursos se hace en proporción de su utilización, basada por ejemplo en el tiempo de ejecución, en lugar del número de recursos asignados.

En la Figura 2.16 se presenta una representación conceptual de una arquitectura sin servidor, la cual ofrece dos conjuntos de servicios: backend como servicio (BaaS, por su definición en inglés) y función como servicio (FaaS, por su definición en inglés) [89]. BaaS se refiere al conjunto de servicios para el manejo del almacenamiento, intercambio de mensajes y bases de datos [146]. Mientras que FaaS

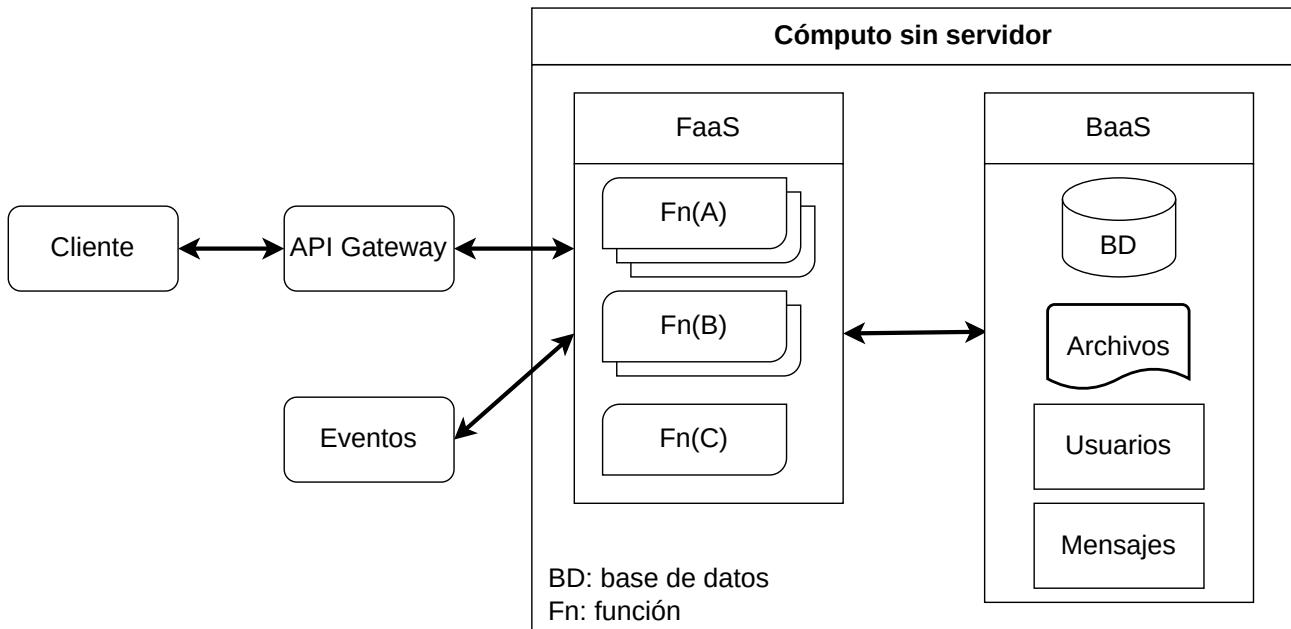


Figura 2.16: Arquitectura sin servidor. Imagen extraída de [89].

está compuesto por un conjunto de servicio que permiten a los desarrolladores ejecutar su propio código (funciones) en plataformas en la nube, el cual es ejecutado con base en un evento especificado por el desarrollador[184]. En este sentido, FaaS hace uso de los servicios de BaaS para manejar los datos producidos por el código desplegado por los desarrolladores, así como el manejo de usuarios e intercambio de mensajes entre las funciones [89, 184].

2.11 Resumen

En este Capítulo se describieron los principales conceptos teóricos relacionados con esta tesis, la cual versa sobre el diseño, despliegue y ejecución de estructuras de procesamiento agnósticas de la infraestructura para el manejo del ciclo de vida de los productos digitales. En este sentido, en este Capítulo se describieron los principales requerimientos no funcionales empleados en el diseño de este método. Además, se presentaron diferentes arquitecturas de software, para comprender sus ventajas y desventajas. En esta tesis, se ha optado por crear soluciones siguiendo una arquitectura de

microservicios, la cual permite diseñar servicios autocontenidos e independientes. Asimismo, en este Capítulo se describieron diferentes patrones de software que pueden ser diseñados en una estructura de procesamiento, así como diferentes tipos de estructuras de procesamiento (flujos de trabajo y tuberías de datos) utilizadas en el manejo y procesamiento de datos para hacer frente a problemas de big data (procesamiento en tiempo real y procesamiento por lotes). Finalmente, en este Capítulo se describieron diferentes infraestructuras para desplegar y manejar estructuras de procesamiento, incluyendo la tecnología de contendores virtuales, la cual es clave en esta tesis para el manejo del despliegue de bloques de construcción.

3

Estado del arte

En el presente Capítulo se presenta un estudio del estado del arte, el cual incluye el trabajo relacionado con el diseño, despliegue y construcción de estructuras de procesamiento para manejar el ciclo de vida de los productos digitales. Al inicio del Capítulo se presenta la descripción de herramientas para crear y manejar arquitecturas de microservicios, las cuales comúnmente son utilizadas para manejar el despliegue y consumo de servicios en la nube. En la segunda parte de este Capítulo se presenta un resumen del manejo de requerimientos no-funcionales con herramientas para crear estructuras de procesamiento. En la siguiente parte del Capítulo se hace una comparación cualitativa de diferentes herramientas para crear soluciones agnósticas. Finalmente, se presenta un análisis de diferentes soluciones utilizadas en el estado del arte para mitigar cuellos de botella en una estructura de procesamiento y manejar el ciclo de vida de las estructuras de procesamiento durante tiempos de diseño, despliegue y ejecución.

Tabla 3.1: Manejo de contenedores virtuales en diferentes herramientas disponibles en la literatura

Trabajo	Ejecución infraestructura				Dependencias de terceros	Adquisición de la imagen de contendor			Nivel de encapsulamiento	Gestión de interfaces de E/S
	MV	CV	Clúster	Una máquina		Archivo TAR	Precargado	Construcción	Estructura de procesamiento	
Pegasus	✓	✓	✓	✓	HTCondor, Docker/Singularity	✓	✓		✓	✓
Makeflow	✓	✓	✓	✓	HTCondor, Docker	✓	✓		✓	
ParSl	✓	✓	✓	✓	Kubernetes		✓			✓
DagOnStar	✓	✓	✓	✓	Docker	✓	✓	✓	✓	✓

3.1 Manejo de contenedores virtuales en la literatura

Para permitir la portabilidad de soluciones, diferentes herramientas del estado del arte han incluido el soporte de contenedores virtuales. En la Tabla 3.1 se muestra un resumen del manejo y uso de contenedores virtuales en Pegasus, Makeflow, ParSl y DagOnStar, los cuales son motores de flujos de trabajo. Como se puede observar, todos los motores requieren de una herramienta de terceros para realizar el manejo de contenedores virtuales (p. ej. Docker o Singularity), mientras que solo Pegasus y Makeflow requieren de una herramienta de terceros (HTCondor) para la gestión de la infraestructura. En el caso de ParSl, este utiliza Kubernetes para realizar la gestión de los contenedores en un clúster de computadoras. Además, en la Tabla 3.1 se puede observar que la gestión de las imágenes de contenedor en cada caso se puede realizar cargando la imagen desde un archivo TAR, utilizando la imagen precargada del sistema o construyéndola a partir de un archivo *Dockerfile*. En este caso, se observó que DagOnStar es la única solución que permite la gestión de las imágenes de las tres maneras, y no impone restricciones en su manejo.

El nivel de encapsulamiento se refiere a si la herramienta permite desplegar toda la estructura de procesamiento en un solo contenedor virtual o si, en cambio, cada etapa de la estructura debe ser desplegada en un contenedor virtual diferente. En el caso de Makeflow, cada etapa es manejada en contenedores virtuales diferentes, los cuales son creados durante el tiempo de ejecución. Mientras que en el caso de ParSl, la estructura de procesamiento completa es desplegada en el mismo contenedor.

En el caso de Pegasus y DagOnStar, proporcionan la opción de elegir el esquema de despliegue.

Finalmente, se observó que DagOnStar es la única solución que implementa interfaces de entrada/salida (E/S) para interactuar con los contendores mediante una API Rest.

3.2 Herramientas para el manejo de microservicios

En la literatura se encuentran disponibles una gran cantidad de herramientas para manejar microservicios durante su diseño, despliegue y ejecución.

En tiempo de diseño, diferentes marcos de trabajo están disponibles para que los programadores desarrollen microservicios. Ejemplos de estos marcos de trabajo incluye Spring Boot [215], Oracle Helidon[154], Molecular [137], y Eclipse Vert.X [163]. Estos marcos de trabajo proveen paquetes a los desarrolladores para escribir microservicios utilizando un lenguaje de programación en específico (p. ej. Java, Python, Golang o NodeJS). Comúnmente, estos marcos de trabajo interponen una limitación que no permite que los microservicios desarrollados con estas herramientas se interconecten con microservicios desarrollados con otras herramientas. Lo anterior es importante en escenarios donde múltiples organizaciones colaboran y comparten servicios o recursos, debido a que puede generar dependencias entre las organizaciones con el software o la plataforma, al estar limitados a utilizar un marco de trabajo en específico.

Durante tiempo de despliegue, herramientas como Istio [84] y Spring Cloud [50] proveen herramientas a los diseñadores para el despliegue de microservicios en la infraestructura disponible (p. ej. en la nube o en clúster de Kubernetes [124]). Istio es un marco de trabajo que incluye módulos para el descubrimiento de servicios en una malla de servicios, la distribución balanceada de servicios, tolerancia a carga, así como para el control de acceso a los servicios basado en autenticación punto-a-punto [84]. La arquitectura de Istio está dividida en dos planos: datos y control (ver Figure 3.1). El plano de datos se encarga de comunicar los servicios que se van a desplegar. Este plano incluye

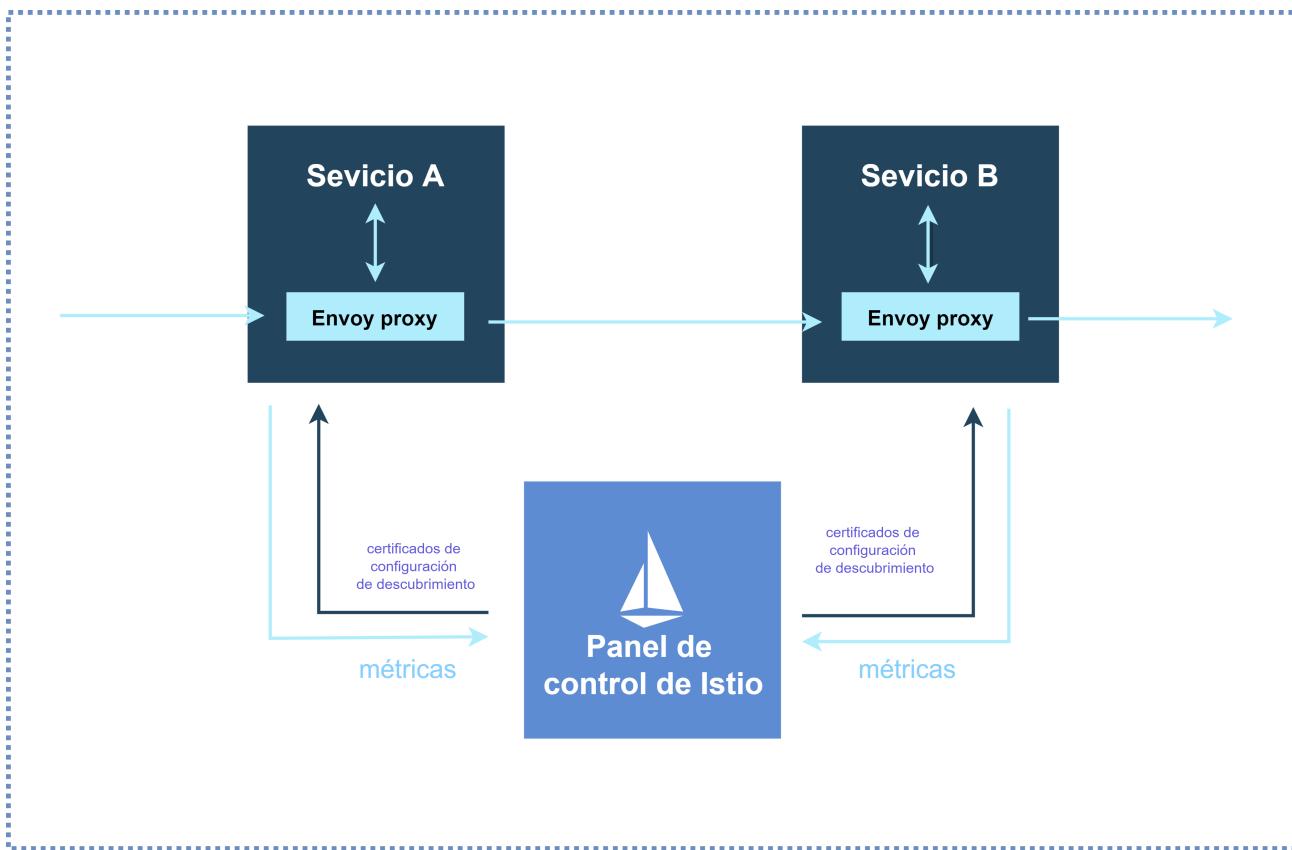


Figura 3.1: Arquitectura de Istio. Imagen extraída de <https://istio.io/latest/about/service-mesh/>.

un conjunto de proxies para enrutar en tráfico entrante a los microservicios. Por su parte, el plano de control se encarga de actualizar las políticas de enrutamiento en los proxies cada que nuevos microservicios sean agregados o se identifiquen cambios en el ambiente de ejecución. Por su parte, Spring Cloud es una herramienta que permite el despliegue en la nube de microservicios desarrollados en Java utilizando la suite de herramientas de Spring. Estos marcos de trabajo y herramientas agilizan el despliegue de microservicios en diferentes plataformas, creando ambientes PaaS, en donde los desarrolladores tienen el control sobre sus aplicaciones y sobre donde son desplegados. Sin embargo, este tipo de despliegue interpone dependencias con las plataformas y el software a utilizar. Por ejemplo, Spring Cloud solo permite el manejo de microservicios desarrollados en Java.

Finalmente, en tiempo de ejecución, herramientas como Kubernetes [124] y Docker Swarm [193] son

utilizadas para realizar la orquestación de los contenedores virtuales y la infraestructura en donde se encuentran desplegados los microservicios. Estas herramientas incluyen módulos para permitir el escalamiento de los servicios en un clúster de computadoras, el manejo del balanceo de carga de los datos de entrada y la tolerancia a falla de los servicios.

3.3 Herramientas para diseñar estructuras de procesamiento

Tradicionalmente los motores de flujos de trabajo han sido ampliamente utilizados para el diseño de estructuras de procesamiento tales como tuberías, patrones o flujos de trabajo. Estos motores de flujos de trabajo permiten a los usuarios finales ejecutar un conjunto de aplicaciones (manejadas como etapas) en diferentes recursos computacionales (principalmente en sistemas distribuidos) [21]. Estos motores comúnmente se encargan de la entrega de los datos a las etapas de la estructura de procesamiento, así como la recolección y visualización de los resultados producidos por cada etapa.

Los motores de flujo de trabajo fueron diseñados originalmente para permitir el diseño, despliegue y ejecución de estudios científicos de gran escala en el *grid* o infraestructuras de cómputo de alto desempeño [13, 23, 123]. En años recientes, diferentes motores de flujo han empezado la transición a soportar otro tipo de infraestructuras como la nube y ambientes contenerizados. Algunos ejemplos de motores de flujo de trabajo populares son Parsl, Makeflow, Pegasus, PyCOMPSs [203] y Globus Galaxy [80].

Taverna [108] es una herramienta utilizada en el modelado y manejo de flujos de trabajo científicos. Taverna permite la construcción de flujos de trabajo utilizando un lenguaje de programación basado en un modelo de flujo de datos, en el cual los flujos de trabajo se diseñan como grafos acíclicos dirigidos, donde los nodos son llamados procesos y representan componentes de software tales como

servicios web o *scripts* [108].

Swift [221] es un lenguaje de *scripting* que permite la construcción de flujos de trabajo paralelos. La sintaxis de Swift está basada en el lenguaje de programación C e implementa funciones de paralelismo y diseño de estructuras guiadas por el flujo de los datos. Swift ejecuta y coordina las etapas/aplicaciones de un flujo de trabajo de manera distribuida en infraestructuras de múltiples procesadores, clústeres de computados, grids y en la nube.

Parsl [16] permite el modelo y desarrollo de flujos de trabajo utilizando el lenguaje de programación Python. Además, Parsl permiten la programación de estructuras de paralelas, las cuales pueden ser ejecutadas de forma paralela en un conjunto de nodos computacionales. Los flujos de trabajo diseñados con este motor pueden ser escalados utilizando patrones paralelos como el patrón Manejador/Trabajador [16].

DagOnStar [139] es un motor de flujos de trabajo basado en Python que permite la ejecución de estructuras paralelas en equipos locales, la nube, clústeres HPC y contenedores Docker. Este motor está diseñado para reducir las tareas de movimiento de datos mediante la implementación de volúmenes compartidos entre todas las etapas de un flujo de trabajo, además implementa un *garbage collector* que se encarga de eliminar aquellos datos que ya han sido procesados y no son requeridos por las etapas del flujo de trabajo.

Pegasus [57] es un motor de flujos de trabajo que permite el diseño de tareas de múltiples pasos utilizadas principalmente en contextos científicos. Pegasus desacopla la ejecución de estas tareas de su entorno. Múltiples aplicaciones pueden ser interconectadas con este motor, manejando cada aplicación como un paso de una tarea. Las tareas pueden ser desplegadas en múltiples infraestructuras en la nube y clústeres locales.

Makeflow [5] es un motor de flujos de trabajo que permite la interconexión de múltiples aplicaciones utilizando un modelo de diseño basado en archivos Make [226]. Este motor permite la creación de

Tabla 3.2: Comparación cualitativa de diferentes herramientas del estado del arte para construir estructuras de procesamiento. ‡ representa que la herramienta utiliza Condor o MPI para agregar paralelismo a las estructuras de procesamiento.

Trabajo	Confiableidad		Eficiencia		Portabilidad		Manejo de la heterogeneidad	Reusabilidad	
	Sistema	Datos	Patrones	Hilos	Flujo de trabajo	Etapas		Servicios	Scripts
Galaxy [80]	✓	-	✓	-	-	-	-	-	✓
Makeflow [226]	✓	-	-	✓‡	✓	-	✓	-	✓
Pegasus [56]	✓	-	-	✓‡	-	✓	✓	-	✓
Parsl [16]	✓	-	✓	✓	✓	✓	✓	-	✓
DagOnStar	✓	-	✓	✓	✓	✓	✓	-	✓
Método propuesto	✓	✓	✓	-	✓	✓	✓	✓	✓

flujos basados en contenedores de las plataformas Docker [28] y Singularity [113]. Los flujos de trabajo desarrollados con Makeflow pueden ser desplegados en múltiples infraestructuras tales como Amazon EC2 y clústeres manejados con HTCondor.

Kulla [171] es un modelo de construcción basado en grafos acíclicos dirigidos y los principios de entrega continua y bloques de construcción encapsulados en contenedores virtuales. Estos contenedores encapsulan las aplicaciones junto a sus dependencias, configuración y variables de entorno. Kulla permite la creación de diferentes estructuras de procesamiento tales como flujos de trabajo, tuberías, patrones de paralelismo y cualquier combinación de estos [171].

3.4 Manejo de requerimientos no funcionales en estructuras de procesamiento

En la Tabla 3.2 se presenta la comparación cualitativa de los requerimientos no funcionales manejados por diferentes herramientas utilizadas comúnmente en el estado del arte para construir diferentes estructuras de procesamiento como tuberías y flujos de trabajo. Para esta comparación se consideraron cinco requerimientos no funcionales, que en esta tesis se consideran claves para crear servicios agnósticos de la infraestructura: confiabilidad, eficiencia, portabilidad, manejo de la

heterogeneidad y reusabilidad.

Confiabilidad se refiere a la capacidad de las estructuras de procesamiento creadas con las herramientas para resistir las diferentes fallas que puedan surgir por fallas en la infraestructura o plataforma donde las etapas se encuentran desplegadas. Esta característica también considera el mantener el acceso a los datos incluso cuando hay fallas en la infraestructura donde estos están almacenados. En esta comparación, todas las herramientas consideradas cuentan con mecanismos de confiabilidad para soportar fallas en las etapas de la estructura de procesamiento, permitiendo recuperar el procesamiento de los datos incluso después de una falla. Por otro lado, solo Kulla y método propuesto en esta tesis incluyen la característica de confiabilidad en datos.

La propiedad de eficiencia es agregada a una estructura de procesamiento en la forma de patrones de paralelismo o hilos paralelos. Como se puede observar, la implementación de hilos es una estrategia común, que en algunas ocasiones (\ddagger en el caso de Pegasus [56] y Makeflow [226]) se requiere llamar a una herramienta de terceros (p. ej. HTCondor [69] y MPI) para proveer paralelismo de tareas, lo cual implica que estas herramientas requieren la instalación de herramientas adicionales para proveer soluciones con paralelismo. En el presente trabajo, en lugar de lanzar un hilo por cada producto a procesar, se hace uso de patrones de paralelismo, en los cuales una etapa de una estructura de procesamiento es clonada n veces, y utilizando técnicas de balanceo de carga, los productos digitales son distribuidos entre los clones. Lo anterior es conveniente cuando se tiene que procesar grandes conjuntos de datos a través de un conjunto de etapas que tienen un desempeño heterogéneo, lo cual puede producir cuellos de botella en el procesamiento de los datos y que son observados por el usuario final en la forma de retrasos en el procesamiento. En este sentido, los patrones reducen estos retrasos mediante el ajuste de sus parámetros (número de trabajadores).

Portabilidad se refiere a la capacidad de poder desplegar una estructura de procesamiento o parte de ella en diferentes infraestructuras. Esto incluye la migración de las estructuras de una infraestructura a otra (por ejemplo, entre diferentes proveedores en la nube). En este sentido, diferentes herramientas

como Parsl, Makeflow y Pegasus permiten la portabilidad de los scripts con los que se desarrollan la estructura de procesamiento, es decir, estos scripts no tienen que ser modificados para permitir la migración de un servicio de la nube a un clúster. En práctica, para desplegar una estructura de procesamiento en diferentes infraestructuras, es común que los desarrolladores realicen tareas (p. ej. modificar el código de las aplicaciones, configuración de variables de entorno o instalar dependencias) para ajustar las etapas y aplicaciones de esta estructura a cada infraestructura [164]. Estas tareas de corrección de errores pueden impactar en el desempeño y funcionamiento de las estructuras de procesamiento debido a malas configuraciones de las etapas o aplicaciones. En este sentido, soluciones como la propuesta en esta tesis y DagOnStar utilizan contenedores virtuales para crear paquetes de software que puedan ser desplegados en múltiples infraestructuras, reduciendo el tiempo de despliegue de las aplicaciones [226].

El manejo de heterogeneidad se refiere a la capacidad de las herramientas para poder interconectar aplicaciones desarrolladas en diferentes lenguajes de programación. Galaxy es la única herramienta que no cumple con esta característica, dado que en esta herramienta las aplicaciones deben de ser desarrolladas como funciones de la misma herramienta o en scripts Linux. El método propuesto en esta tesis no solo permite manejar la heterogeneidad con la que son desarrolladas las aplicaciones, además permite manejar la heterogeneidad en la infraestructura utilizada para desplegar las aplicaciones. Lo anterior, hace posible la interconexión de aplicaciones desplegadas en múltiples infraestructuras como en el borde, la niebla, la nube o clústeres HPC.

La propiedad de reusabilidad posibilita la reutilización de servicios, aplicaciones y estructuras de procesamiento previamente desarrolladas en una nueva solución. En este sentido, en el estado del arte se pueden identificar dos enfoques de reusabilidad: de scripts y de servicios. En el primero, las herramientas permiten que los scripts, configuraciones o código de una estructura de procesamiento puedan ser reutilizados para crear una nueva. Como se puede observar, la mayoría de las herramientas evaluadas siguen este enfoque. En el segundo enfoque, las herramientas permiten a las organizaciones

	Makeflow	Parsl	funcX	Parsl+funcX	DagOnStar	Kulla	Método propuesto
Nativo	(E) FT	(E) Io	(RU) FT	(RU) E FT (Io)	(RU) E Io (FT)	(P) FT (RU) Io	(RU) R E (C) I A (FT) Io
Utilizando herramientas externas	(E) []			(P)	(P)		(P)
Parcialmente	(P) RU	(P) RU	(P)				
(E) Eficiente (Io) Interoperabilidad (P) Portabilidad (FT) Tolerancia a fallos (RU) Reusabilidad (R) Confidabilidad de datos (C) Confidencialidad (A) Control de Acceso (I) Integridad							
<i>Ambiente de ejecución</i> ◆ No distribuido ■ Distribuido en un solo ambiente ▲ Distribuido en múltiples ambientes							

Figura 3.2: Comparación del manejo de requerimientos no funcionales en diferentes soluciones del estado del arte.

y diseñadores reutilizar servicios previamente desplegados para agregarlos a una nueva estructura de procesamiento. Este enfoque es el seguido por el modelo propuesto y permite reducir el tiempo empleado en desplegar servicios y aplicaciones que ya han sido desplegados anteriormente.

En la Figura 3.2 se muestra un resumen de los diferentes requerimientos no funcionales manejados por diferentes herramientas del estado del arte, incluyendo el método propuesto en esta tesis. Las herramientas se categorizaron de acuerdo con al cumplimiento de los requerimientos previamente descritos: eficiencia, manejo de la heterogeneidad, reusabilidad, confiabilidad y seguridad. En el caso de la confiabilidad se evaluó desde dos perspectivas:

- Tolerancia a fallos sistemática, es decir, cuando un componente de la herramienta o aplicación falla, el sistema es capaz de recuperarse del fallo de forma transparente para el usuario.

- En caso de que una parte de la infraestructura de almacenamiento donde los datos son almacenados no se encuentre disponible, el usuario debe de ser capaz de acceder a sus datos. Este tipo de confiabilidad comúnmente es cumplimentado con algoritmos de codificación de datos (p. ej. el algoritmo de dispersión de información) [167].

Para cumplimentar esta comparación, se consideró el cumplimiento de requerimientos de seguridad para el manejo de datos, tales como: confidencialidad, integridad y control de acceso. Estos requerimientos son cruciales cuando se manejan ciclos de vida de productos digitales sensibles, tales como datos médicos.

Las herramientas consideradas en la Figura 3.2 son Makeflow [226], Parsl [16], funcX [42], Parsl+funcX, DagOnStar [180] y Kulla [171]. En cada herramienta se comparó si los requerimientos se cumplimentan de forma nativa, utilizando herramientas externas o de forma parcial. Además, se incluye una evaluación si las herramientas permiten la construcción de estructuras de procesamiento para entornos no distribuidos, un solo ambiente distribuido (p. ej. la nube) o múltiples ambientes distribuidos (p. ej. una combinación de la nube y la niebla).

Como se puede observar, el método propuesto en esta tesis es la única herramienta que cumple con la mayoría de los requerimientos no funcionales evaluados de forma nativa y en ambientes distribuidos. La única excepción es la portabilidad, en la cual se utiliza un gestor de contenedores para crear paquetes de software que puedan ser desplegados en múltiples infraestructuras.

3.5 Soluciones agnósticas para manejar el ciclo de vida de los productos digitales

En la Tabla 3.3 muestra un resumen de diferentes marcos de trabajo y herramientas clasificados de acuerdo a su enfoque para crear estructuras de procesamiento (orientadas al diseño o al despliegue)

Tabla 3.3: Comparación cualitativa de soluciones agnósticas para manejar el ciclo de vida de los productos digitales.

Trabajo	Enfoque		Nivel de agnosticismo			Acoplamiento y ejecución
	Orientado al diseño	Orientado al despliegue	(1)	(2)	(3)	
Taverna [108]	✓	-	✓	-	-	Centralizado
Comps [48]	✓	-	✓	✓	✓	Centralizado
Jenkins [26]	-	✓	✓	✓	-	Centralizado
Globus galaxies [125]	✓	-	✓	-	-	Centralizado
Parsl [16]	✓	-	✓	✓	-	Centralizado
Makeflow [226]	✓	-	✓	-	-	Centralizado
Pegasus [56]	✓	-	✓	-	-	Centralizado
DagOnStar [180]	✓	-	✓	✓	-	Centralizado
Istio [84]	-	✓	✓	✓	-	Autocontenido
OpenShift [36]	-	✓	✓	-	-	Centralizado
Slurm [100]	-	✓	✓	-	-	Centralizado
HTCondor [69]	-	✓	✓	-	-	Centralizado

(1) Agnóstico al lenguaje; (2) Agnóstico a la plataforma; (3) Agnóstico a la infraestructura.

y a su nivel de agnosticismo (al software, plataforma o infraestructura).

Las soluciones orientadas al diseño son aquellas en donde las aplicaciones son acopladas a alto nivel, comúnmente utilizando una interfaz gráfica, modelos de programación o esquemas declarativos [201]. Por ejemplo, Parsl y DagOnStar proveen a los usuarios finales un modelo de programación basado en Python, que abstrae a los usuarios del despliegue de las aplicaciones y el manejo de los datos. En estas herramientas, los diseñadores deben de crear un DAG a partir que incluya las etapas de su estructura de procesamiento. De forma similar, Makeflow y Pegasus también proveen interfaces basadas en archivos de configuración para crear DAGs que representan una estructura de procesamiento.

Por otra parte, las soluciones orientadas al despliegue se encargan de realizar la calendarización y distribución de las etapas y aplicaciones en una estructura de procesamiento a la infraestructura donde serán desplegadas. En este sentido, estas soluciones incluyen calendarizadores (p. ej. HTCondor), gestores de colas (p. ej. Slurm) y APIs de proveedores en la nube (p. ej. AWS). Estas soluciones comúnmente son mezcladas con una solución orientada al diseño, las cuales delegan el despliegue de tareas y el manejo de los recursos de infraestructura y datos, lo cual termina creando una dependencia con estas. Por ejemplo, Makeflow y DagOnStar delegan el despliegue de aplicaciones en clústeres HPC a herramientas como HTCondor[69]/Slurm[100]. Las soluciones orientadas al despliegue son

desarrolladas para funcionar en una infraestructura o plataforma en específico (p. ej. Slurm y HTCondor solo están disponibles para clústeres Linux). Esto hace que la migración de las estructuras de procesamiento generadas con estas soluciones sea costosa y complicada [153].

En la presente tesis se propone la integración de solución orientada al diseño y al despliegue en una sola solución autocontenido. De esta manera, se mantiene a alto nivel el diseño de las estructuras de procesamiento, y las etapas de esta estructura al ser autocontenidoas podrán desplegarse y manejarse por sí mismas.

El nivel de agnosticismo de una solución se puede evaluar en tres niveles dependiendo de la dependencia que se mitigue o elimine: agnóstico al software [30], agnóstico a la infraestructura [171] o agnóstico a la plataforma [94]. Es importante destacar, que una solución agnóstica a la plataforma es también agnóstica del software, y, por lo tanto, una solución agnóstica a la infraestructura también lo es a la plataforma y el software.

Las soluciones agnósticas del software, como Pegasus y Makeflow permiten la interconexión de múltiples aplicaciones heterogéneas desarrolladas utilizando diferentes lenguajes de programación [30]. Estas soluciones eliminan la dependencia de las organizaciones de tener que desarrollar las aplicaciones, utilizadas en un ciclo de vida, con el mismo lenguaje de programación. En cambio, las organizaciones pueden reutilizar aplicaciones previamente probadas e interconectarlas para crear estructuras de procesamiento, sin importar el lenguaje de programación en el que hayan sido desarrolladas. Sin embargo, comúnmente estas soluciones utilizan herramientas de terceros para manejar el despliegue de las aplicaciones, la infraestructura y el manejo de los datos.

Las soluciones agnósticas de la plataforma permiten la creación de estructuras de procesamiento portables, las cuales comúnmente son desplegadas en un solo ambiente (por ejemplo, en la nube o un clúster HPC). Estas soluciones rompen las dependencias que puede generar una estructura de procesamiento con las plataformas utilizadas para el despliegue y manejo de aplicaciones. Ejemplos de este tipo de soluciones son DagOnStar e Istio [84], las cuales no utilizan herramientas de terceros para

desplegar aplicaciones en la infraestructura disponible. Sin embargo, las estructuras de procesamiento o servicios creados con estas soluciones no pueden desplegarse en múltiples infraestructuras, lo cual hace complicado la interconexión de la infraestructura de diferentes organizaciones o crear sistemas jerárquicos distribuidos como el borde-niebla-nube. En este sentido, uno de los objetivos de las soluciones agnósticas de la infraestructura es el permitir la interconexión de diferentes infraestructuras, creando flujos de datos entre estas [171]. Además, las estructuras de procesamiento y los servicios agnósticos de la infraestructura tienen la capacidad de ser desplegados en múltiples infraestructuras aun sin conocer sus características de hardware y software. Un ejemplo de una solución agnóstica de la infraestructura es Comps [48], el cual permite crear flujos de datos a través de clústeres HPC y la nube.

Comúnmente en la literatura las herramientas que siguen el acoplamiento y ejecución centralizado en el cual existe un componente central que se encarga de manejar el despliegue, acoplamiento y ejecución. Una ventaja de este enfoque es que permite que se reutilice una estructura de procesamiento cuando esté desplegada y lista para ejecutarse. Sin embargo, este enfoque tiene la desventaja que, al centralizar estos procesos, se tiene un único punto de falla, lo cual en caso de este componente falle detendría el procesamiento de los datos. En cambio, en un enfoque autocontenido, cada componente de una estructura de procesamiento contiene todo lo necesario para desplegarse y funcionar automáticamente en múltiples infraestructuras, así como para autoacoplarse con el resto de los componentes de la estructura de procesamiento. Además, cada componente está a cargo de mover los datos que requiera procesar, y en caso de que un componente falle, el resto puede seguir funcionando.

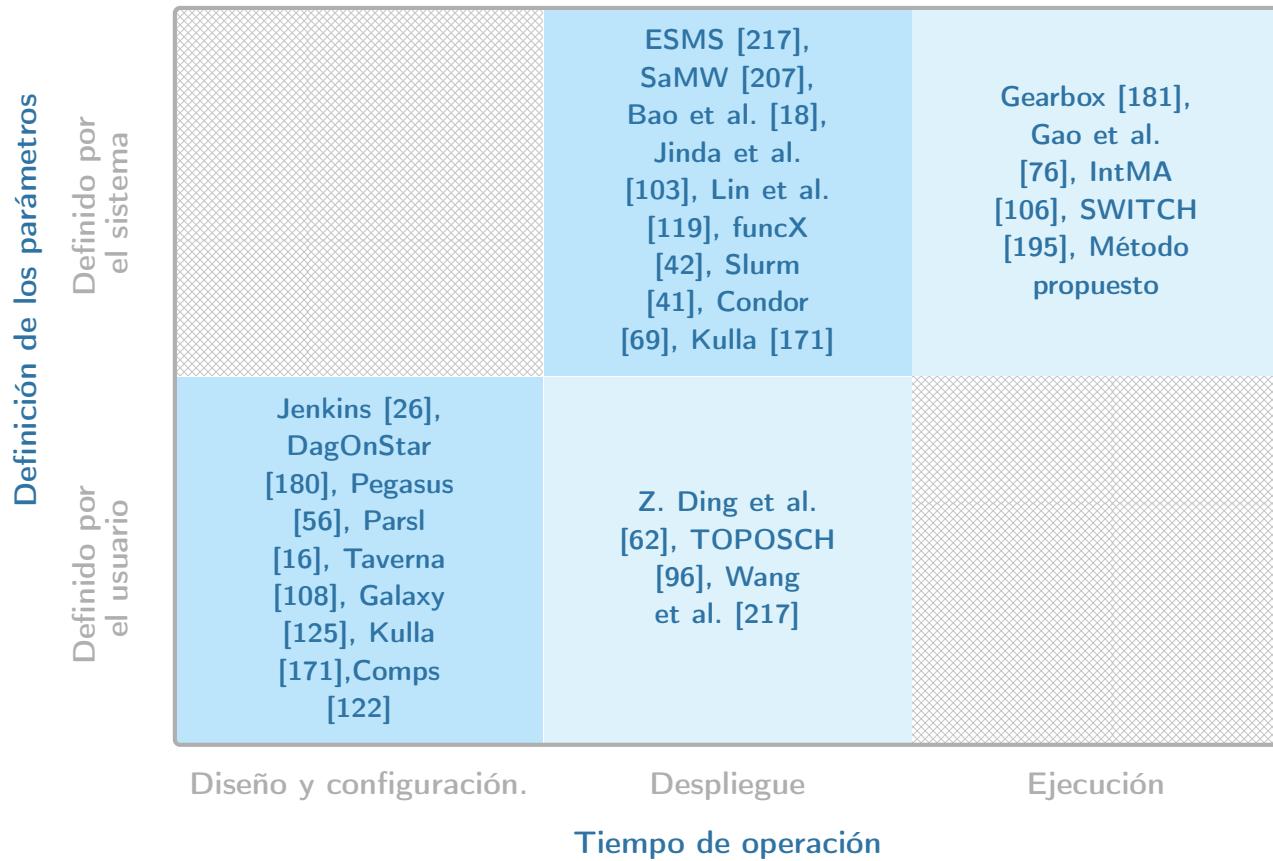


Figura 3.3: Clasificación de herramientas del estado del arte que implementan técnicas para mitigar cuellos de botella. *Definido por el usuario*: el usuario especifica los parámetros para mitigar los cuellos de botella. *Definidos por el sistema*: el sistema identifica y decide como mitigar los cuellos de botella.

3.6 Manejo de cuello de botellas en estructuras de procesamiento

Tradicionalmente, las herramientas utilizadas para la creación de estructuras de procesamiento implementan esquemas para mejorar la eficiencia en el procesamiento de datos, y si es identificado poder mitigar cuellos de botella. En la Figura 3.3 se muestra una clasificación de diferentes soluciones que implementan este tipo de esquemas. Primero, las soluciones se clasificaron de acuerdo con el tiempo en el que las soluciones mitigan los cuellos de botella: durante el diseño y configuración de

la estructura de procesamiento, durante su despliegue o durante su ejecución. Posteriormente, las soluciones fueron clasificadas de acuerdo con cómo se identifican los cuellos de botella: *i*) si son identificados por el usuario y este es quien define los parámetros de eficiencia (paralelismo, balanceo de carga o recursos computacionales) para mitigar el cuello de botella y *ii*) si el sistema identifica automáticamente el cuello de botella e implementa una técnica para mitigarlo.

En tiempo de diseño y configuración, diferentes herramientas y soluciones, como motores de flujos de trabajo (DagOnStar, Parsl y PyComps) y gestores de tuberías (Nextflow), permiten a los desarrolladores/usuarios definir distintos parámetros de eficiencia para mejorar el rendimiento de sus estructuras de procesamiento o de un cuello de botella en específico. Por ejemplo, indicando el número de trabajadores/hilos de cada etapa en la estructura, eligiendo como balancear la carga entre los trabajadores, seleccionar la infraestructura/plataforma en donde se ejecutará cada etapa o aumentar/limitar la memoria disponible para cada etapa. Sin embargo, para identificar y mitigar un cuello de botella en tiempo de diseño y configuración, el diseñador requiere de conocimiento previo sobre las aplicaciones manejadas en la estructura de procesamiento, así como su desempeño en la infraestructura disponible. Lo anterior requiere de una etapa de pruebas y experimentación que puede ser costosa tanto en términos monetarios como en tiempo.

Durante tiempo de despliegue los diseñadores pueden crear archivos de configuración para declarar en qué infraestructura será desplegada cada aplicación. Por ejemplo, soluciones como DagOnStar [180], TOPOSCH[96], Kulla [171] y Jenkins [12] permiten a los diseñadores la declaración del nodo, equipo, máquina virtual o contenedor virtual donde cada etapa de una estructura de procesamiento será desplegada. En este caso, los diseñadores pueden elegir que aquellas etapas que se consideren cuellos sean desplegadas en equipos con mayor poder de cómputo. De nuevo, para poder hacer esta planeación, los diseñadores requieren tener conocimiento previo sobre el desempeño de las aplicaciones que serán desplegadas en cada etapa.

En tiempo de despliegue existen soluciones en las que el sistema automáticamente elige la

infraestructura donde será desplegada cada aplicación considerada en cada etapa de una estructura de procesamiento, así como los parámetros de estas aplicaciones. Soluciones como Slurm[41], funcX[42], Condor[69] y ESMS [217] implementan algoritmos de calendarización para desplegar las aplicaciones en una piscina de recursos computacionales siguiendo un conjunto de restricciones (p. ej. presupuesto, latencia o requerimientos de calidad de servicio). Algunas de estas soluciones implementan algoritmos de optimización para predecir el desempeño de las aplicaciones y producir una configuración a partir de esta predicción [18, 103, 119]. Con estas soluciones los desarrolladores delegan a un sistema la selección de la infraestructura donde sus aplicaciones serán desplegadas. Sin embargo, estas soluciones requieren tener un conocimiento previo sobre el desempeño de las aplicaciones (comúnmente entregado a los algoritmos en forma de trazas) para entrenar los algoritmos que implementan.

En este contexto, se han desarrollado diferentes soluciones para identificar y mitigar, en tiempo de ejecución, cuellos de botella en las estructuras de procesamiento. La principal ventaja de estas soluciones es que implementan técnicas que no requieren la intervención de los usuarios para mejorar el desempeño de las aplicaciones. Por ejemplo, Gearbox [181], SWITCH [195] e IntMA [106] implementan algoritmos para identificar cuellos de botella en el procesamiento de datos y mitigarlos, implementando técnicas de paralelismo, manejo de memoria y reducción de la latencia entre aplicaciones. SWITCH [195] implementa un sistema de monitoreo para identificar el estado de las aplicaciones y realizar el autoesacalamiento de aquellas aplicaciones computacionalmente costosas. IntMA [106] implementa un algoritmo que se encarga de agrupar durante tiempo de ejecución las aplicaciones desplegadas con el objetivo de reducir la latencia entre estas, la cual puede producir cuellos de botella en el intercambio de datos. Gearbox [181] es una herramienta que implementa esquemas de monitoreo para recolectar datos de desempeño de las etapas en una tubería e identificar cuellos de botella para mitigarlos, implementando técnicas de paralelismo basadas en patrones y técnicas de manejo de memoria.

El método propuesto en esta tesis implementa un esquema declarativo en el que, en tiempo de diseño, los diseñadores pueden elegir el número de trabajadores y la técnica de balanceo de carga de cada etapa en la estructura de procesamiento. En tiempo de ejecución, este método incluye un modelo para la identificación y mitigación de cuellos de botella basado en un esquema de manejo de colas y autoescalamiento. La principal ventaja de este método con otros propuestos en el estado del arte es que este modelo permite interconectar a un alto nivel múltiples infraestructuras desplegadas en múltiples infraestructuras (p. ej. en el borde, la niebla o la nube). Además, en caso de que haya cambios en la composición de las estructuras de procesamiento, el modelo de identificación y mitigación de cuellos de botella se adapta a estos cambios para seguir mitigando cuellos de botella en esta nueva composición.

3.7 Discusión

El método propuesto permite el manejo del ciclo de vida de las estructuras de procesamiento, creando soluciones agnósticas que son independientes de la infraestructura. En este sentido, en esta Sección se presenta un análisis del manejo del ciclo de vida de las estructuras de procesamiento utilizando diferentes herramientas disponibles en el estado del arte. Además, en la segunda parte de esta sección se presenta un análisis del grado de independencia de algunas de estas herramientas.

3.7.1 Manejo del ciclo de vida de las estructuras de procesamiento

En la Tabla 3.4 se presenta un resumen de diferentes soluciones utilizadas para construir y manejar estructuras de procesamiento durante su ciclo de vida. Esta tabla incluye tuberías de software y big data, una interfaz de programación paralela, herramientas para procesamiento en tiempo real, tuberías CI/CD y flujos de trabajo.

Tabla 3.4: Herramientas identificadas en la literatura para construir y manejar el ciclo de vida de las estructuras de procesamiento.

Trabajo	Enfoque principal	Modelo			Comportamiento		TSP	Paralelismo			Escalamiento		
		Proc.	Prog.	Com.	Estático	Dinámico		Dis.	Des.	Eje.	Hilos	Patrones	vertical
GRPPI [58]	IntPP.	✓	✓	-	✓	-	✓	-	-	-	✓	✓	-
Kulla [171]	Tubs.	✓	✓	✓	✓	-	✓	-	-	-	✓	✓	✓
Gearbox [181]	Tubs.	✓	-	✓	-	✓	✓	-	A	-	✓	✓	-
Apache Flink [37]	ProTR.	✓	✓	-	✓	-	✓	-	-	✓	-	✓	✓
Sacbe [83]	Tubs.	✓	✓	✓	✓	-	✓	-	-	-	-	✓	-
RaftLib [25]	ProTR.	-	✓	✓	✓	-	✓	-	A	✓	-	✓	-
Jenkins [12]	Tubs. CI/CD	✓	✓	✓	✓	-	✓	-	-	✓	-	✓	✓
FastFlow [6]	PatP.	✓	✓	✓	✓	-	✓	-	-	-	✓	✓	-
Nextflow [60]	Tubs.	✓	✓	✓	✓	-	✓	-	-	-	✓†	✓	✓*
DagOnStar [180]	FT.	✓	✓	✓	✓	-	✓	-	-	✓	✓	✓	✓*
Parsl [16]	FT.	✓	✓	✓	✓	-	✓	-	-	✓	✓	✓	✓*
Makeflow [5]	FT.	✓	✓	✓	✓	-	✓	-	-	✓	-	✓	✓*
C-Stream [177]	ProTR.	✓	✓	✓	-	✓	✓	✓	✓	A	✓	-	✓
SPar [86]	ProTR.	✓	✓	✓	-	✓	✓	✓	✓	A	✓	-	✓
Método propuesto	EstPA.	✓	✓	-	✓	✓	✓	✓	✓	A	-	✓	✓

†: solo tuberías.

*: utilizando herramientas de terceros.

A: automático.

Proc.: Procesamiento

Prog.: Programación

Com.: Comunicación

Dis.: Diseño

Des.: Despliegue

Eje.: Ejecución

TSP: Tiempo de sintonización de parámetros

IntPP.: Interfaz de programación paralela.

Tubs.: Tuberías.

ProTR.: Procesamiento en tiempo real.

Tubs. CI/CD: Tuberías CI/CD.

PatP.: Patrones de paralelismo.

FT.: Flujos de trabajo.

EstPA.: Estructuras de procesamiento agnósticas.

Soluciones como Kulla, Sacbe, Nextflow y Makeflow permiten construir sistemas interconectando bloques de construcción en contenedores, donde cada bloque encapsula una aplicación para gestionar y procesar los datos entrantes y producir información útil. Los contenedores virtuales añaden la propiedad de portabilidad a los sistemas, lo que resulta útil en entornos borde-niebla-nube para desplegar una aplicación a través de diferentes infraestructuras.

En tiempo de diseño, los modelos de programación permiten la generación de estructuras de procesamiento mediante código. Por ejemplo, Parsl [16] y DagOnStar [180] proveen un modelo de programación basado en Python para crear flujos de trabajo paralelos basados en hilos, permitiendo el encadenamiento de aplicaciones heterogéneas. De forma similar, Spark [178] incluye un modelo de programación basado en Scala [107] para crear aplicación de big data para el procesamiento de datos en memoria. Mientras que [12] y Nextflow [60] proveen un modelo de programación basado en Groovy [19] para crear tuberías.

En tiempo de despliegue y ejecución, el modelo de comunicación permite la comunicación de las

aplicaciones consideradas en la estructura de procesamiento. Makeflow, por ejemplo, permite el acoplamiento de aplicaciones mediante la declaración de las entradas y salidas de cada aplicación, y durante tiempo de ejecución el intercambio de datos se realiza utilizando el sistema de archivos. Además, Makeflow puede ser conectado con herramientas de terceros como MPI [22] para manejar el paso de mensajes entre aplicaciones. De forma similar, Nextflow crea canales de comunicación para intercambiar los datos, creando referencias a los datos en el sistema de archivos.

Comúnmente las estrategias de paralelismo son agregadas a las estructuras de procesamiento para mejorar su desempeño. Herramientas como C-Stream, Makeflow, Jenkins y Apache Flink implementan estrategias de paralelismo basadas en hilos. Mientras, herramientas como Kulla, Gearbox, Apache Spark y FastFlow implementan patrones genéricos para manejar los datos en paralelo. Ejemplos de estos patrones incluyen el patrón gestor/trabajador, divide&vencerás, patrones de tuberías, map/reduce y fork/join. Sin embargo, estos patrones suelen ser estáticos y no pueden modificarse en tiempo de ejecución (por ejemplo, para ajustar el número de trabajadores en los patrones). Estos cambios son cruciales cuando surgen eventos durante tiempo de ejecución que requieren adaptar la estructura de procesamiento a las nuevas condiciones de la infraestructura, la carga de trabajo entrante o el entorno de ejecución. Por ejemplo, durante eventos de interrupciones en la infraestructura, cuellos de botella en la estructura de procesamiento o cambios en el tamaño de la carga de trabajo. Herramientas centradas en el diseño y despliegue de estructuras de procesamiento (por ejemplo, Jenkins, Kulla, Parsl o Spark), el gestor primero tiene que detener la ejecución de las estructuras de procesamiento y desplegar una nueva versión de la estructura de procesamiento con los cambios requeridos. Esto es crucial en escenarios donde se requiere la operación continua de las aplicaciones en la estructura de procesamiento (p. ej. procesamiento en tiempo real de datos de sensores o la entrega de datos médicos), debido a que los datos arriban continuamente para ser procesados en tiempo real.

El ajuste de parámetros en una estructura o patrón de procesamiento puede realizarse en tiempos

de diseño, despliegue y ejecución. Durante tiempo de diseño, los diseñadores crean una estructura de procesamiento definiendo sus parámetros (por ejemplo, el número de hilos/trabajadores o el tamaño de los búferes), aplicaciones, entradas/salidas y patrones de acoplamiento. Durante tiempo de despliegue, herramientas como Makeflow y Jenkins programan el despliegue de las estructuras de procesamiento con base en los recursos disponibles. En este sentido, los diseñadores pueden definir un conjunto de restricciones basadas en un presupuesto (cuando se trabaja en la nube pública) o requisitos de hardware (p. ej. memoria, núcleos o disco). Durante tiempo de ejecución, herramientas como Gearbox, Raftlib, C-Stream, FastFlow y Spar, ajustan automáticamente las estructuras de procesamiento y sus componentes para mejorar su rendimiento y mitigar los cuellos de botella. Por ejemplo, C-Stream modifica el grado de paralelismo añadiendo o eliminando hilos durante tiempo de ejecución. Por su parte, Raftlib [25] minimiza la sobrecarga de comunicación y gestiona el buffer de las aplicaciones dinámicamente con técnicas de aprendizaje automático. Por su parte, FastFlow [6] es un marco de trabajo que permite el diseño de diferentes patrones paralelos para la creación de soluciones paralelas, incluyendo la sincronización y comunicación de sus componentes. Vogel et al. [214] presentan una estrategia autoadaptativa que amplía FastFlow para cambiar dinámicamente entre diferentes patrones (por ejemplo, etapas de canalización, etapas paralelas, etc.) para satisfacer un requerimiento de calidad de servicio (QoS, por su definición en inglés). Por otro lado, Gearbox mitiga los cuellos de botella en una tubería, creando dinámicamente nuevos trabajadores para las aplicaciones con mayores tiempos de respuesta y gestionando el intercambio de datos mediante un esquema en memoria. SPar se ha ampliado en diferentes trabajos para crear sistemas autoadaptativos que dinámicamente pueden ajustar el grado de paralelismo en las etapas, creando soluciones de autoescalamiento basadas en las fluctuaciones del rendimiento de las etapas y los requisitos de QoS del diseñador. Estas herramientas (Gearbox, Raftlib, C-Stream, FastFlow y SPar) crean sistemas autoadaptativos que pueden adaptar el comportamiento de un sistema basándose en diferentes estrategias (aumentar el número de trabajadores o cambiar entre patrones) y requisitos (por ejemplo, rendimiento o latencia). Sin embargo, herramientas como Gearbox, FastFlow y Spear no están

preparadas para trabajar en entornos distribuidos donde los componentes de un patrón o una estructura de procesamiento pueden estar desplegados en diferentes máquinas.

Las estructuras de procesamiento creadas con las herramientas presentadas en la Tabla 3.4 pueden producir un comportamiento estático o dinámico durante tiempo de ejecución. Las estructuras con un comportamiento estático son aquellas estructuras que no cambian la conformación de su diseño, aplicaciones o parámetros durante tiempo de ejecución. Mientras que las estructuras con un comportamiento dinámico son aquellas que durante tiempo de ejecución controlan y modifican la estructura de procesamiento para producir un nuevo comportamiento.

El método propuesto en la presente tesis está enfocado en la creación y manejo de estructuras de procesamiento agnósticas de procesamiento durante su ciclo de vida (diseño, despliegue y ejecución). El objetivo es mitigar las dependencias con el lenguaje de programación, la plataforma y la infraestructura mediante la implementación de tres modelos. El primero es un modelo de construcción basado en bloques autosimilares y autocontenidos, el cual durante tiempo de diseño permite el acoplamiento de diferentes aplicaciones y diseño de diferentes patrones, mitigando la dependencia con el lenguaje de programación. El segundo es un modelo de programación y comunicación que permite el despliegue de las estructuras de procesamiento en múltiples infraestructuras, permitiendo que los patrones puedan escalar tanto vertical como horizontalmente. Estos modelos mitigan la dependencia de las estructuras de procesamiento con los gestores de orquestación de datos y tareas. Finalmente, el tercero es un modelo gestión de tareas/datos y paralelismo implícito que permite mitigar los cuellos de botella en la estructura de procesamiento, durante tiempo de ejecución. Este modelo genera un comportamiento dinámico en las estructuras de procesamiento, debido a que implementa un esquema de monitoreo y gestión de colas utilizando estrategias de autoparalelismo. En este sentido, la sintonización de parámetros en el método propuesto puede realizarse de forma manual durante tiempos de diseño y despliegue, y de forma manual durante tiempo de ejecución.

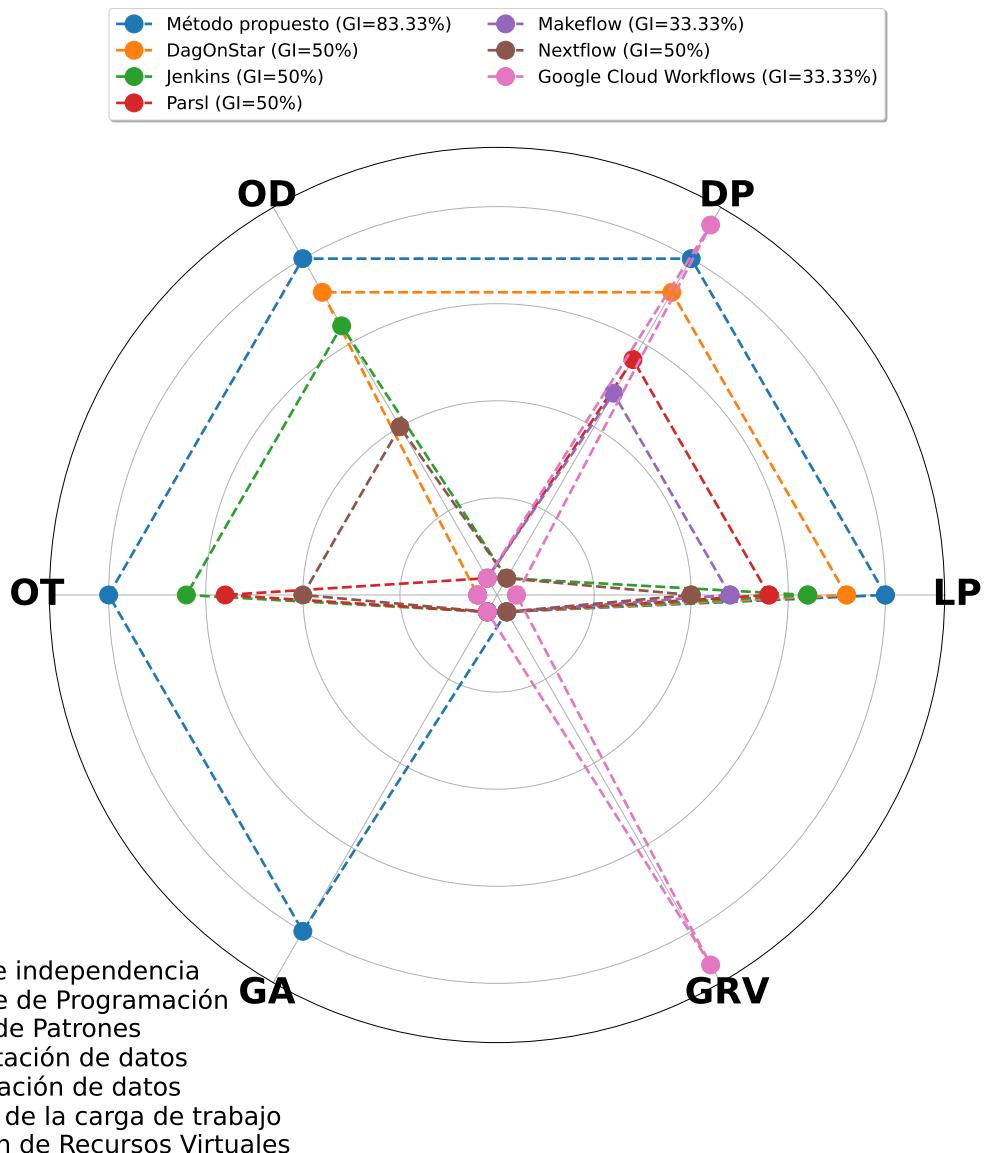


Figura 3.4: Grado de independencia de diferentes herramientas del estado del arte.

3.7.2 Análisis de independencia de herramientas para el manejo de estructuras de procesamiento

El método propuesto en esta tesis busca reducir la dependencia de las estructuras utilizadas en un CVPD con el lenguaje de programación, la plataforma y la infraestructura, al aplicar cada modelo

durante tiempos de diseño, despliegue y ejecución. En este sentido, y a modo de comparación con soluciones del estado del arte, en la Figura 3.4 se muestra una comparativa cualitativa del grado de independencia alcanzado por las estructuras de procesamiento desarrolladas utilizando los siguientes marcos de trabajo del estado del arte: Google Cloud Workflows [85], DagOnStar, Jenkins, Parsl, Makeflow y Nextflow, así como el método propuesto. Este grado de independencia fue obtenido al evaluar si los marcos de trabajo permiten, durante tiempo de diseño, el manejo de la heterogeneidad en el lenguaje de programación y el diseño de patrones, en tiempo de despliegue, la orquestación de datos y tareas, y durante tiempo de ejecución la gestión de la carga de trabajo y de los recursos virtuales disponibles. En este sentido, se espera que una solución completamente agnóstica de la infraestructura tenga un grado de independencia (*GI*) del 100 %. Como se puede observar, el método propuesto tiene un *GI* de 83.33 % debido a que permite gestionar la mayoría de los aspectos evaluados, con la excepción de la gestión de recursos virtuales debido a que para crear los bloques de construcción de este método se utiliza la plataforma de contenedores Docker [28] para agilizar el despliegue de las aplicaciones.

4

Método de procesamiento continuo para la construcción y manejo de estructuras de procesamiento agnósticas de la infraestructura

En este Capítulo se describe un método de procesamiento continuo que agrega la propiedad de agnosticismo a las estructuras de procesamiento utilizadas para manejar un conjunto de aplicaciones utilizadas durante el ciclo de vida de los productos digitales (CPVD). Una estructura de procesamiento se define mediante un conjunto de reglas que permiten la integración de múltiples aplicaciones requeridas en un CPVD. El método propuesto permite el manejo de las estructuras de procesamiento durante tiempos de diseño, despliegue y ejecución. En tiempo de diseño, las estructuras de procesamiento son creadas utilizando un modelo de construcción basado en bloques autosimilares y autocontenidos. Este modelo permite a los diseñadores acoplar un conjunto heterogéneo de

aplicaciones eliminando la dependencia de las estructuras de procesamiento con el lenguaje de programación. En tiempo de despliegue, un modelo de comunicación y programación permite el instanciamiento de las aplicaciones en la infraestructura de cómputo disponible. Como resultado, la estructura de procesamiento es presentada a los usuarios finales como un sistema de aplicaciones para el manejo de CVPDs. Este modelo mitiga la dependencia de las estructuras de procesamiento con las plataformas y la infraestructura. Finalmente, durante tiempo de ejecución, un modelo de gestión de tareas/datos y paralelismo implícito permite mitigar los cuellos de botella que pueden surgir en una estructura de procesamiento y que pueden afectar la experiencia del usuario final al retrasar la entrega de los resultados producidos durante el CVPD.

En este sentido, en este Capítulo se describen los siguientes modelos:

1. Un modelo de construcción que permite encapsular las aplicaciones consideradas en una estructura de procesamiento para manejar un CVPD en bloques autosimilares y autocontenidos.
2. Un modelo de comunicación y programación de infraestructuras como código (IaC) para materializar y desplegar, en forma transparente y automática, estructuras de procesamiento agnósticas sobre múltiples infraestructuras de cómputo.
3. Un modelo de gestión de tareas/datos y paralelismo implícito que mitiga los efectos de los cuellos de botella en una estructura de procesamiento. Este modelo incluye un esquema de monitoreo para identificar cuellos de botella en la estructura de procesamiento y mitigarlos utilizando técnicas de manejo de colas y autoparalelismo.

Además, en este Capítulo se presenta la arquitectura general del método propuesto, la cual permite el diseño, despliegue y ejecución de estructuras agnósticas de la infraestructura.

4.1 Modelo de construcción de estructuras de procesamiento basadas en bloques autosimilares y autocontenidos

En esta sección se describen los principios de diseño de *PuzzleMesh*, un modelo para construir estructuras de procesamiento que soporten el manejo del ciclo de vida de los productos digitales, y desplegarlas como servicios agnósticos de la infraestructura. Este modelo está inspirado en una metáfora de rompecabezas, en donde diferentes piezas independientes son unidas para formar una pieza más grande llamada rompecabezas (solución). En este sentido, en este modelo una aplicación puede ser vista como una pieza de un rompecabezas, que al ser unida con otras aplicaciones (piezas) formarán una solución mayor. Un rompecabezas en el caso de las piezas, y una estructura de procesamiento en el caso de las aplicaciones. A diferencia de los juegos de rompecabezas tradicionales, donde cada pieza tiene un número limitado de interconexiones de entrada y salida, y que sólo puede ser interconectada con un número limitado de piezas, en nuestro modelo las aplicaciones pueden tener un número ilimitado de interconexiones de entrada y salida, permitiendo su interconexión con cualquier aplicación disponible (similar a los rompecabezas basados en bloques de construcción).

4.1.1 Manejo de aplicaciones como piezas de rompecabezas

Siguiendo esta metáfora de rompecabezas en *PuzzleMesh*, una aplicación es manejada como una pieza que básicamente representa un artefacto de software que incluye diferentes componentes para garantizar el correcto despliegue y ejecución de esta aplicación, generando un paquete de software autocontenido listo para desplegar y ejecutar en diferentes infraestructuras sin hacer cambios en la configuración ni en el código fuente de la aplicación. La Figura 4.1 muestra la representación de la arquitectura en pila de una pieza, así como su representación conceptual como una pieza de un juego

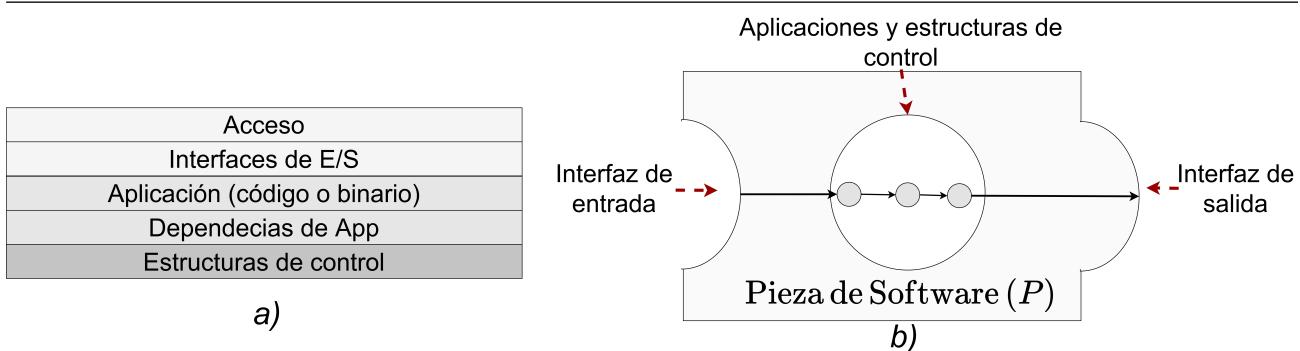


Figura 4.1: Diseño de una pieza de rompecabezas en PuzzleMesh: a) Pila arquitectónica de sus componentes y b) su representación conceptual en la metáfora de rompecabezas.

de rompecabezas tradicional. Las capas consideradas en la pila arquitectural de la Figura 4.1a son las siguientes:

Capa de acceso. Verifica que el acceso a la pieza y sus componentes sea válido. Para lo anterior, se utiliza un sistema de tokenización, en el cual cada usuario y aplicación cuenta con un conjunto de tokenes que son validados en esta capa.

Interfaces de entrada y salida (E/S). Permiten la lectura y escritura de datos utilizando la red, el sistema de archivos o la memoria principal. Las interfaces de entrada leen los contenidos desde una fuente de datos, y las interfaces de salida escriben los resultados o datos producidos por una pieza en un destino de datos. Como se puede observar en la Figura 4.1b, las interfaces de entrada y salida representan la cuenca y curvatura de una pieza de rompecabezas real, respectivamente.

Aplicación. Código fuente o binario de la aplicación que será manejada como una pieza.

Metadatos. Incluye los parámetros de entrada y salida de la aplicación, así como la ruta y el comando de ejecución para invocar el código fuente o binario de la aplicación.

Dependencias de la aplicación. Incluye las librerías y variables de entorno requeridas por la aplicación para ser ejecutada en la misma manera que fue probada y aprobada para pasar

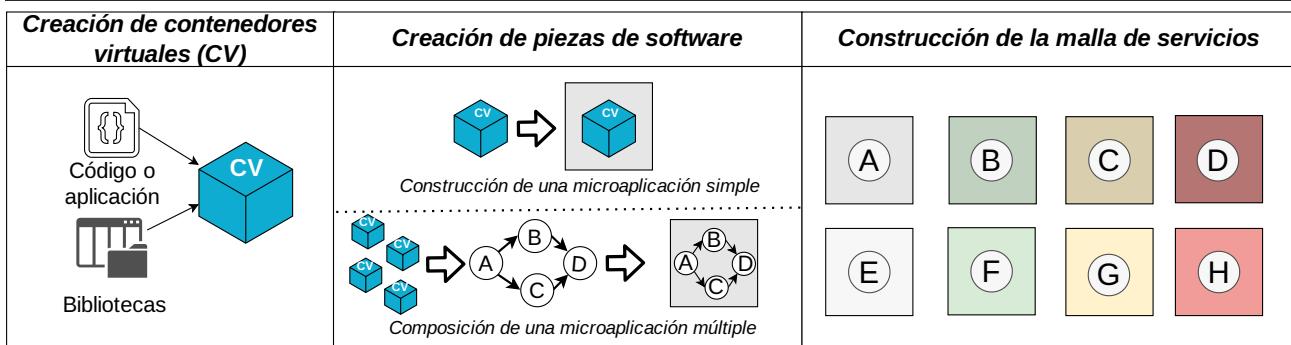


Figura 4.2: Proceso de diseño y manejo de piezas de software en PuzzleMesh.

a producción.

Estructuras de control y manejo. Controlan la ejecución de estructuras agregadas a una pieza para mejorar su eficiencia, tales como patrones de paralelismo, balanceadores de carga y herramientas de manejo de datos.

En la Figura 4.1b se muestra como esta pila es convertida en una pieza durante tiempo de ejecución, la cual es manejada como una caja negra e incluye los componentes descritos. El objetivo es abstraer el manejo de aplicaciones como piezas que permitan la construcción orientada al diseño de estructuras de procesamiento mediante el acoplamiento de bloques autosimilares y autocontenidos.

En la Figura 4.2 se muestra el proceso de diseño y manejo de piezas de software en PuzzleMesh. El proceso inicia con la encapsulación de una aplicación y todos sus componentes y dependencias en un contenedor virtual (CV). El siguiente paso es diseñar una pieza de software siguiendo dos esquemas de composición:

Composición simple. Diseño de una pieza de software compuesta solamente por un contenedor virtual, el cual contiene una aplicación para el procesamiento de datos.

Composición múltiple. Diseño de una pieza de software mediante el encadenamiento de múltiples contenedores virtuales.

La configuración de una pieza se realiza mediante un esquema declarativo en el cual los diseñadores

deben agregar cada contenedor virtual que desea manejar con dicha pieza de software. En este esquema declarativo los diseñadores crean una pieza mediante la descripción de sus componentes (fuentes de datos, funcionalidad, imagen de contenedor virtual y destino de datos) en un archivo de configuración (ver un ejemplo en el Anexo B).

Finalmente, el proceso de diseño de una pieza termina cuando ésta es agregada a una malla de servicios, la cual se encarga del almacenamiento e indexamiento de todas las piezas construidas por los diseñadores y usuarios finales.

4.1.2 Acoplamiento de piezas de software para crear estructuras de procesamiento

En PuzzleMesh, la unión de un conjunto de piezas de software crea un rompecabezas que básicamente representa una estructura de procesamiento (p. ej. un flujo de trabajo o una tubería) que permite el manejo del ciclo de vida de los productos digitales. El orden de ejecución de las piezas y su unión son definidos durante tiempo de diseño mediante un grafo acíclico dirigido (DAG, por sus siglas en inglés), el cual determina cómo los datos son intercambiados entre las piezas utilizando sus interfaces de E/S. El inicio de un DAG es una o múltiples fuentes de datos que contienen los datos en crudo que al ser procesados a través de un rompecabezas producirán productos digitales. Por otra parte, los nodos finales del DAG son destinos de datos, los cuales representan el destino de los datos (p. ej. equipos de almacenamiento o los usuarios finales).

En este sentido, un rompecabezas es conveniente para manejar el ciclo de vida de los productos digitales. En tiempo de despliegue, al instanciar un rompecabezas se crea un sistema de aplicaciones para el manejo de CVPDs. Para producir este sistema de aplicaciones, un rompecabezas implementa los siguientes componentes:

Arquitectura de microservicios. Permite el manejo de las piezas consideradas en el

rompecabezas como servicios verticales que pueden ser consumidos por los usuarios finales y otras aplicaciones.

Metadatos. Contienen las rutas de acceso a las fuentes y destinos de datos, así como el DAG que establece el orden de ejecución de las piezas.

API REST. Permite que los usuarios consuman los resultados producidos por un rompecabezas, así como el estado de este.

Manejo de almacenamiento. Un rompecabezas incluye una red de distribución de contenidos para manejar la entrega de datos entre las piezas, así como el almacenamiento de datos y productos digitales. Esta red está basada en un modelo de publicación/suscripción, lo cual permite diseñar diferentes patrones de intercambio y compartición de datos, lo cual permite sincronizar los datos intercambiados entre las piezas del rompecabezas.

Estos componentes se encuentran autocontenido en un rompecabezas, lo cual significa que tanto las piezas como los rompecabezas manejan el despliegue de sus componentes, así como su acoplamiento con otros componentes. En PuzzleMesh, es posible reutilizar rompecabezas previamente creados para unirlos y crear una nueva estructura de software llamada meta-rompecabezas, la cual implementa un componente para coordinar la ejecución de los rompecabezas considerados en este meta-rompecabezas. Esta construcción de estructuras de procesamiento basada en piezas, rompecabezas y meta-rompecabezas crea estructuras autosimilares, las cuales permiten el manejo del ciclo de vida de los productos digitales a diferentes niveles (aplicación, sistema y servicio).

4.1.3 Modelando estructuras de procesamiento agnósticas de la infraestructura como un rompecabezas

En esta sección se presenta un modelo para formalizar el diseño de estructuras de procesamiento como rompecabezas y desplegarlos como servicios agnósticos de la infraestructura. Para modelar

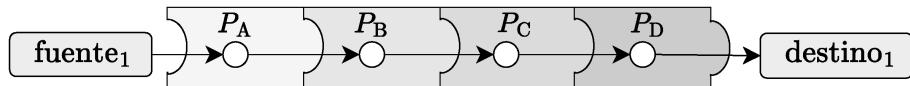


Figura 4.3: Representación conceptual de un rompecabezas construido en la forma de una tubería.

una pieza (P), primero se define una funcionalidad (F) que puede estar compuesta por una sola aplicación (composición simple) o múltiples aplicaciones (composición múltiple) organizadas en la forma de una tubería. Por lo tanto, esta funcionalidad se puede definir de la siguiente manera:

$$F = \begin{cases} A_1, & \text{Composición simple} \\ A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n, & \text{Composición múltiple} \end{cases} \quad (4.1)$$

donde n es el número de aplicaciones consideradas para ser manejadas.

Esta funcionalidad F es agregada a una pieza (P) junto con un conjunto de interfaces de entrada (E) y salida (S), las dependencias de las aplicaciones y un conjunto de estructuras de control para la creación de piezas de software autocontenidas. Una pieza puede modelarse como a continuación:

$$P = [E, F, S]. \quad (4.2)$$

donde F es la funcionalidad de la pieza de software, E y S son las interfaces de entrada y salida que pueden ser implementadas utilizando la red, memoria o el sistema de archivos. Esto puede ser representado de la siguiente manera:

$$(E, S) = \begin{cases} \text{red} \\ \text{memoria} \\ \text{sistema_de_archivos} \end{cases}. \quad (4.3)$$

La interconexión de dos o más piezas genera un rompecabezas, el cual básicamente es una estructura

de procesamiento para el manejo de datos y productos digitales. En la Figura 4.3 se muestra una tubería construida mediante el acoplamiento de cuatro piezas (P_A, P_B, P_C y P_D) conectadas a una fuente y un destino de datos. El rompecabezas procesa los datos de la fuente y los deposita en el destino.

Como se mencionó anteriormente, un rompecabezas es diseñado como un DAG mediante la interconexión de las interfaces de entrada y salida de un conjunto de piezas de software. Los nodos de este DAG representan las piezas utilizadas para diseñar el rompecabezas, y los vértices son las dependencias de datos que existen entre estas piezas. La interconexión de dos o más piezas para formar un rompecabezas se realiza utilizando sus interfaces de entrada y salida. En PuzzleMesh, esta interconexión es denotada de la siguiente manera:

$$P_i \rightarrow P_{i+1}. \quad (4.4)$$

Esto representa que P_{i+1} consumirá, utilizando su interfaz de entrada, los datos entregados por la interfaz de salida de P_i . Por lo tanto, un rompecabezas (R) diseñado en la forma de un DAG, el cual es representado de la siguiente manera:

$$R = (\mathbf{N}, \mathbf{V}), \quad (4.5)$$

donde N es un conjunto de nodos representados por las piezas y fuentes/destinos de datos disponibles, mientras que V representa la interconexión de nodos (vértices), lo cual puede ser denotado de la siguiente manera:

$$E = [\text{fuente}_x \rightarrow P_1], [P_1 \rightarrow P_2], [P_2 \rightarrow P_3], \dots, [P_{n-1} \rightarrow P_n], [P_n \rightarrow \text{destino}_y], \quad (4.6)$$

donde fuente_x es la fuente de datos con identificador x , n es el total de piezas en el rompecabezas R y destino_y es el destino de datos con identificador y . En este rompecabezas, el proceso inicia

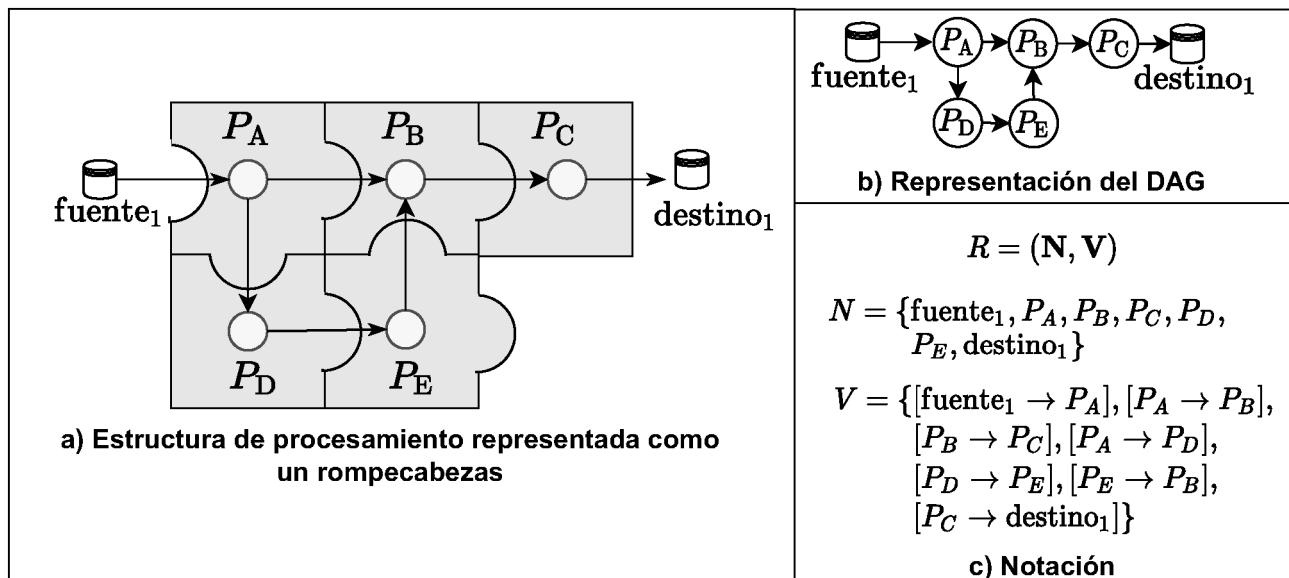


Figura 4.4: Representación conceptual de a) un rompecabezas creado mediante el acoplamiento de piezas, b) su representación como un DAG y c) y su notación en PuzzleMesh.

cuando P_1 adquiere productos digitales desde fuente $_x$, y finaliza cuando P_n escribe los resultados del procesamiento en destino $_y$. En la Figura 4.4 se presenta un ejemplo de un rompecabezas diseñado con cinco piezas, así como su representación como DAG y su notación.

4.1.4 Esquemas de preparación de datos para el manejo de requerimientos no funcionales en productos digitales

Los ciclos de vida para el manejo de datos sensibles (p. ej. datos médicos) requieren el cumplimiento de un conjunto de requerimientos no funcionales, tales como confidencialidad, integridad, control de acceso y confiabilidad. Esto es importante en escenarios donde los datos son compartidos a través de canales no controlados (p. ej. Internet) o almacenados y procesados utilizando infraestructura de terceros (p. ej. la nube pública). Estos requerimientos no funcionales son agregados a los datos utilizando aplicaciones específicas para su cumplimiento. Por ejemplo, el algoritmo de dispersión de información (IDA, por sus siglas en inglés) permite agregar la característica de confiabilidad a los

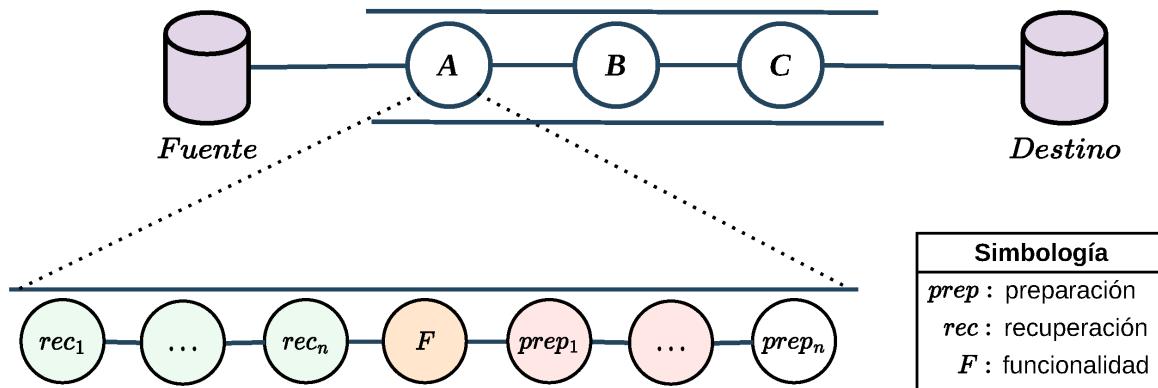


Figura 4.5: Ejemplo de un esquema de recuperación y preparación de datos incluidos en las interfaces de entrada y salida de una pieza.

datos para tolerar fallos en la infraestructura donde estos son almacenados [38]. Otro ejemplo es el esquema de cifrado por bloques *Advanced Encryption Standard* (AES) [52], el cual permite agregar la característica de confidencialidad a los datos.

En PuzzleMesh, las aplicaciones para agregar requerimientos no funcionales a los datos son organizadas en la forma de tuberías, siguiendo los mismos principios que la construcción de una pieza de software P . Estas tuberías son agregadas en las interfaces de entrada y salida de las piezas, creando dos tipos de esquemas de manejo de datos:

Esquemas de preparación de datos. Son agregados en las interfaces de salidas de las piezas e implementan aplicaciones para cumplimentar requerimientos no funcionales en los datos. Por ejemplo, cifrando los datos con AES.

Esquemas de recuperación de datos. Son agregados en las interfaces de entrada de las piezas e implementan aplicaciones para recuperar los datos originales que fueron procesados con los esquemas de preparación. Por ejemplo, si los datos fueron cifrados con AES utilizando una llave de 256 bits, el esquema de recuperación debe de incluir una aplicación que permita descifrar los datos utilizando esta llave.

En la Figura 4.5 se presenta la representación gráfica de cómo estos esquemas son agregados en

las interfaces de entrada y salida de las piezas. Estos esquemas pueden contener tantas aplicaciones sean requeridas para cumplimentar un conjunto de requerimientos no funcionales.

4.2 Modelo de comunicación y programación de infraestructuras como código

En tiempo de diseño, un rompecabezas es creado siguiendo un modelo de programación basado en un esquema declarativo (ver Anexo B para un ejemplo de un archivo de configuración de un rompecabezas). Este modelo de programación permite a los diseñadores crear, a alto nivel, estructuras de procesamiento mediante la interconexión de diferentes aplicaciones manejadas como rompecabezas. En tiempos de despliegue y ejecución, el modelo de comunicación incluido en las piezas permite su acoplamiento, así como el intercambio de datos y mensajes entre estas piezas. Este modelo de comunicación está basado en REST APIs para el intercambio de mensajes entre piezas y rompecabezas, así como la implementación de una red de distribución de contenidos para realizar el intercambio de productos digitales. Este modelo de comunicación crea un flujo de datos y un canal de comunicación entre las piezas de un rompecabezas.

Estos modelos de comunicación y programación permiten manejar la infraestructura disponible como código, mediante la declaración de las piezas y rompecabezas disponibles en múltiples infraestructuras. Para manejar a alto nivel un conjunto de rompecabezas, y permitir su interconexión con otros rompecabezas desplegados en diferentes infraestructuras, se diseñó una abstracción llamada *meta-rompecabezas* (Ω). La cual permite mitigar, en tiempo de despliegue, las dependencias entre el diseño de un rompecabezas y la plataforma/infraestructura disponible, permitiendo la creación de flujos de datos a través de múltiples plataformas e infraestructuras. En este sentido, un meta-rompecabezas se crea mediante un esquema declarativo, los componentes del sistema (meta-rompecabezas) así como la infraestructura donde se desplegará, permitiendo gestionar esta

infraestructura como código.

La creación de un meta-rompecabezas es conveniente cuando se desean crear flujos de datos que permitan el intercambio de datos entre departamentos de una organización (flujos intraorganizacionales) o incluso entre diferentes organizaciones (flujos interorganizacionales), así como la compartición de recursos de software e infraestructura.

El proceso de creación de un meta-rompecabezas es similar al de un rompecabezas. Dos rompecabezas se pueden acoplar utilizando las interfaces de entrada y salida de sus piezas, lo cual se denota de la siguiente manera:

$$(P_1 \in R_1) \Rightarrow (P_2 \in R_2), \quad (4.7)$$

lo que significa que la pieza P_1 del rompecabezas R_1 está acoplada con la pieza P_2 del rompecabezas R_2 , permitiendo el intercambio de datos entre los rompecabezas R_1 y R_2 . Formalmente, un meta-rompecabezas está compuesto por un conjunto de fuentes, destinos y piezas, de la siguiente manera:

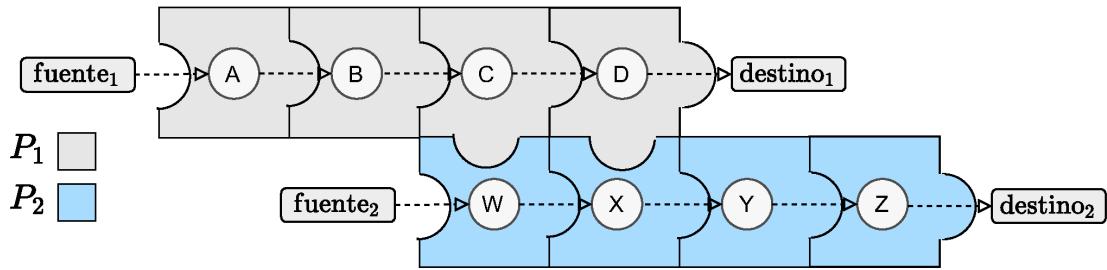
$$\Omega = \{\mathbf{FD}, \mathbf{int}, \mathbf{DD}\}, \quad (4.8)$$

donde **FD** es un conjunto de tamaño n que contiene las fuentes de datos disponibles ($\mathbf{FD} = \{\text{fuente}_1, \text{fuente}_2, \dots, \text{fuente}_n\}$), **DD** es un conjunto de tamaño m que contiene los destinos de datos ($\mathbf{DD} = \{\text{destino}_1, \text{destino}_2, \dots, \text{destino}_n\}$) e **int** es un conjunto que contiene las interconexiones entre piezas de distintos rompecabezas para formar el meta-rompecabezas, lo cual es denotado de la siguiente manera:

$$\mathbf{int} = \{[(P_1 \in R_1) \Rightarrow (P_2 \in R_2)], \dots, [(P_a \in R_x) \Rightarrow (P_b \in R_y)]\} \quad (4.9)$$

donde x e y son los identificadores de dos rompecabezas acoplados utilizando las piezas a y b .

En la Figura 4.6 se muestra una representación conceptual de un meta-rompecabezas integrado por



$$\begin{aligned}\Omega &= \{\text{FD, int, DD}\} \\ \text{FD} &= \{fuente_1, fuente_2\} \\ \text{DD} &= \{destino_1, destino_2\} \\ \text{int} &= \{[(C \in P_1) \Rightarrow (W \in P_2)], [(D \in P_1) \Rightarrow (X \in P_2)]\}\end{aligned}$$

Figura 4.6: Representación conceptual de un meta-rompecabezas compuesto de dos rompecabezas.

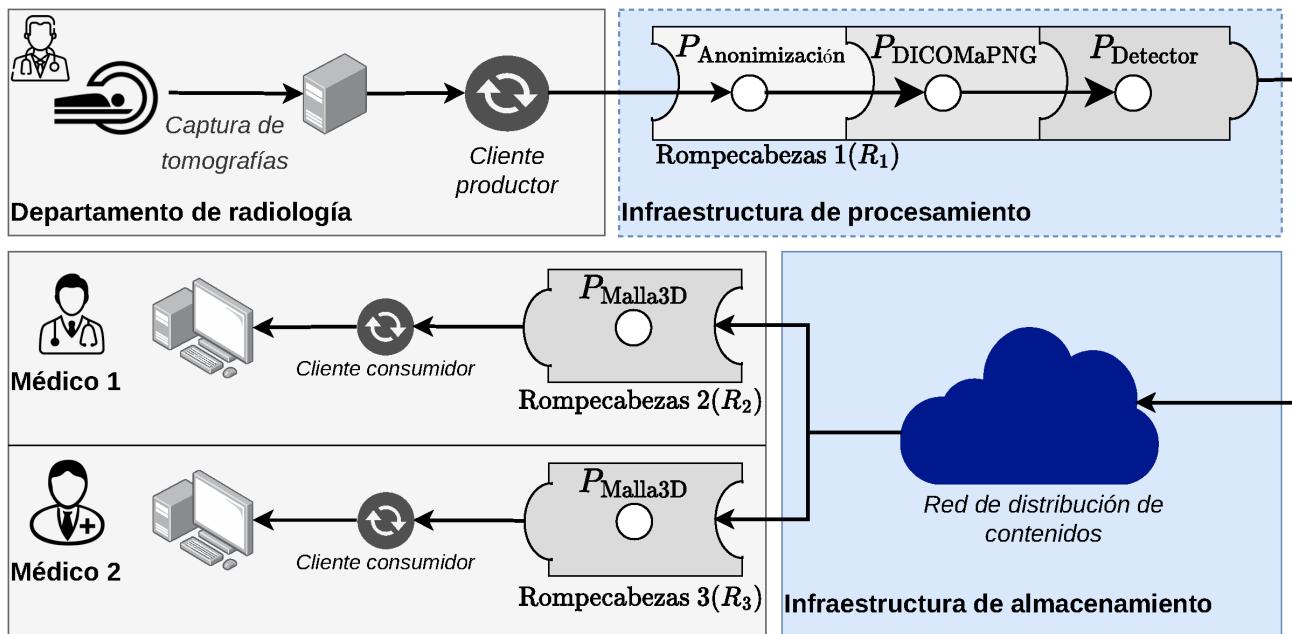


Figura 4.7: Ejemplo de un flujo de datos intraorganizacional en un hospital.

dos rompecabezas (R_1 y R_2), los cuales se encuentran unidos mediante el acoplamiento de las piezas P_C y P_D de R_1 con P_W y P_X de R_2 .

4.2.1 Flujos de datos intraorganizacionales

Un flujo de datos intraorganizacional es creado cuando un meta-rompecabezas interconecta múltiples usuarios dentro de una misma organización. Estos flujos automáticamente procesan, entregan y recuperan datos a los miembros autorizados para acceder a los datos y resultados en una organización. En este sentido, para automatizar la entrega y recuperación de estos datos, los usuarios finales tienen acceso a un cliente, el cual puede realizar operaciones de carga de datos y la recuperación de resultados. Este cliente puede tomar dos roles:

Productor. Un cliente productor carga los datos o productos digitales para ser procesados a través del meta-rompecabezas.

Consumidor. Adquiere los resultados producidos por el meta-rompecabezas.

En la Figura 4.7 se ilustra un ejemplo de un flujo de datos intraorganizacional para el manejo de datos médicos un hospital. En este flujo, un técnico radiólogo captura imágenes de tomografía de un paciente, las cuales son almacenadas en una computadora ubicada en el departamento de radiología. Los datos son adquiridos por el *cliente productor* y procesados a través de un meta-rompecabezas que considera dos rompecabezas:

1. El primero para la identificación automática de tumores en las radiografías. Para este fin, este rompecabezas incluye tres piezas para la anonimización de las tomografías eliminando datos personales de los pacientes, la conversión de las imágenes de formato DICOM a PNG y finalmente una red neuronal convolucional para la identificación de tumores en las imágenes. Este rompecabezas se despliega en un servidor en las instalaciones del hospital. La última pieza en el rompecabezas prepara los resultados agregándoles requerimientos no funcionales tales como confidencialidad utilizando el algoritmo de AES-256 [141] y confiabilidad utilizando el algoritmo de dispersión de información [167]. En este sentido, estos algoritmos realizan las operaciones de cifrado y codificación sobre los datos, respectivamente. Los datos preparados

son entregados a una red de distribución de contenidos para su almacenamiento.

2. El segundo rompecabezas toma de entrada los datos almacenados en la red de distribución de contenidos para su procesamiento con una pieza que realiza la generación de una representación en malla 3D de las tomografías, para que el médico las pueda visualizar. Por lo tanto, la pieza en su interfaz de entrada recupera los datos aplicando las operaciones inversas (decodificación y descifrado) que se les aplicaron a los datos durante su preparación, con el objetivo de que los datos puedan ser procesados por esta pieza. Este rompecabezas se despliega en la computadora de los médicos interesados en consumir los resultados (p. ej. oncólogos), los cuales pueden acceder a los resultados a través de un *cliente consumidor*.

4.2.2 Flujos de datos interorganizacionales

Los flujos de datos interorganizacionales permiten la interconexión de usuarios pertenecientes a múltiples organizaciones. Las metas de estos servicios son:

1. Permitir que los usuarios de diferentes organizaciones puedan compartir datos.
2. Permitir que las organizaciones puedan compartir recursos de procesamiento y almacenamiento de datos para manejar datos de forma distribuida.

En la Figura 4.8 se muestra un ejemplo de un flujo de datos interorganizacional para el manejo de tomografías adquiridas por un técnico radiólogo en un Hospital A. Utilizando un cliente productor, el radiólogo entrega los datos a un rompecabezas el cual procesa los datos a través de las piezas de anonimización, conversión de DICOM a PNG y una red neuronal para la identificación de tumores en las tomografías. Los resultados son preparados y entregados a la red de distribución de contenidos, desde donde los hospitales B y C recuperan los datos. En la infraestructura de estos hospitales, se encuentra desplegado un rompecabezas para la recuperación de los datos y su procesamiento con una pieza para la generación de una malla 3D de las tomografías. Finalmente, utilizando un cliente

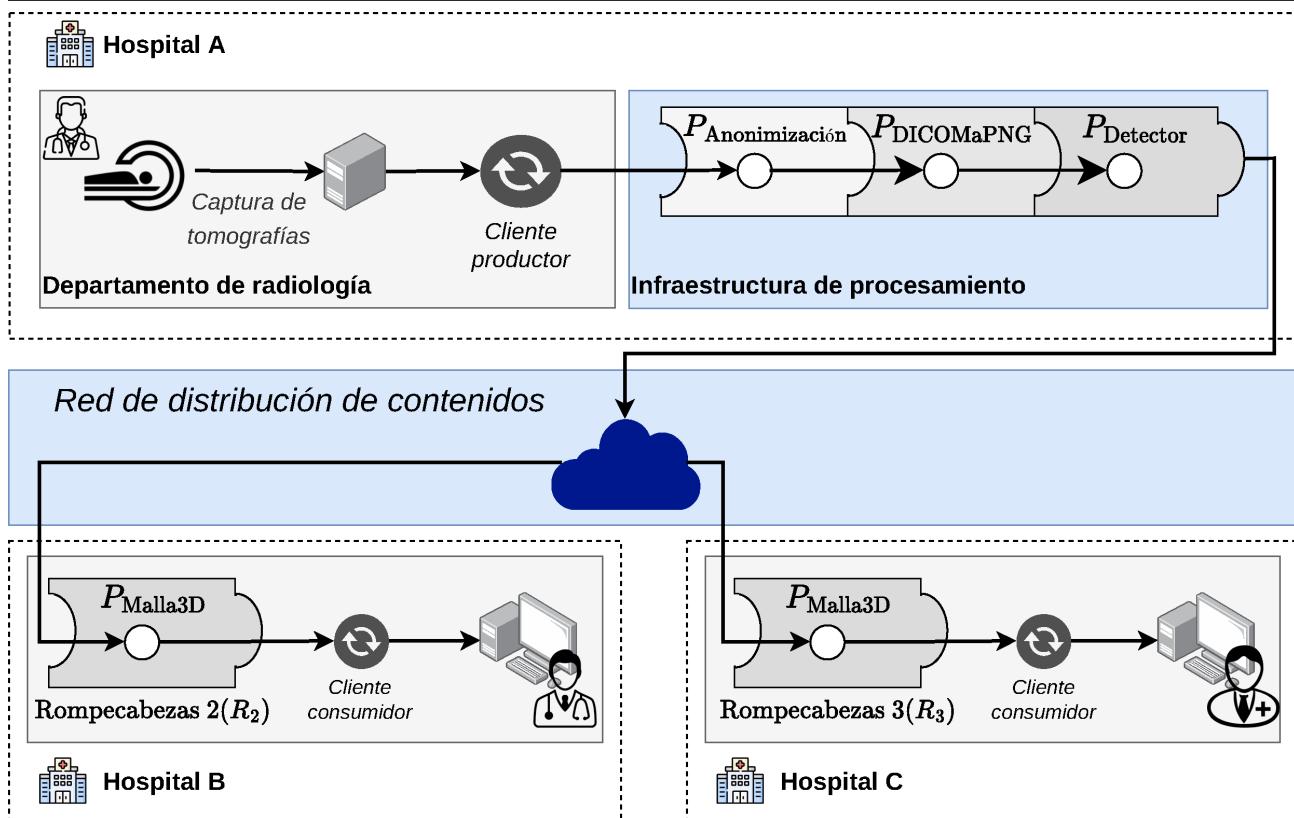


Figura 4.8: Ejemplo de un flujo de datos interorganizacional construido con PuzzleMesh para el intercambio de datos entre hospitales.

consumidor, los médicos de estos hospitales pueden acceder a los resultados.

4.2.3 Malla de servicios para el manejo de servicios

Los meta-rompecabezas, rompecabezas y piezas son incorporados en una malla de servicios (Ψ), que además incluye las fuentes y destinos de datos. Esta malla produce un repositorio desde el cual las organizaciones pueden seleccionar las piezas requeridas para crear servicios para manejar sus productos digitales. En PuzzleMesh, el repositorio de servicios puede ser consultado por los desarrolladores mediante un conjunto de APIs de programación disponibles en la malla de servicios.

Esta malla incluye mecanismos de control de acceso para solo permitir el acceso a los servicios a usuarios autorizados, con el objetivo de que estos puedan modificar y reutilizar servicios previamente

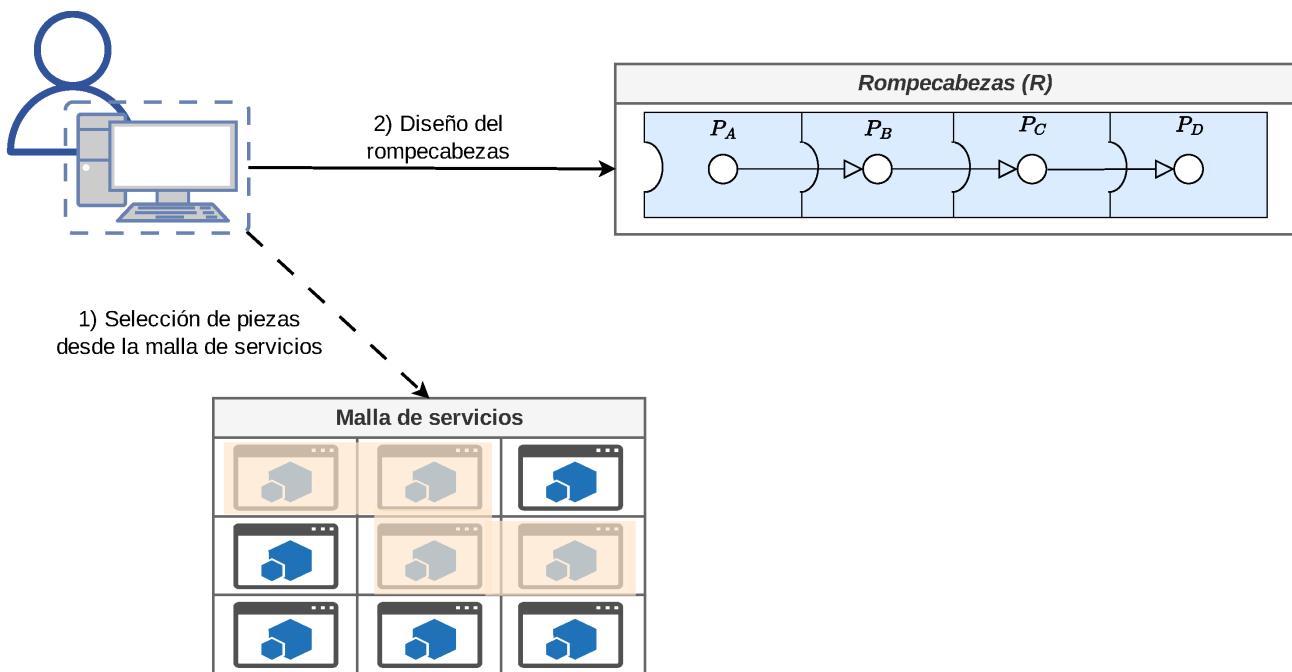


Figura 4.9: Representación conceptual del repositorio de piezas disponibles en la malla de servicios.

existentes. Además, la malla de servicios incluye un mecanismo de paso de mensajes basado en un esquema de publicación/suscripción permite sincronizar los metadatos y datos entre las piezas de un rompecabezas.

En la Figura 4.9 se muestra la representación conceptual de la construcción de un rompecabezas a partir de piezas disponibles en la malla de servicios, la cual permite a los diseñadores y organizaciones reutilizar rompecabezas y piezas creados dentro de la organización o incluso creados por otras organizaciones cuando estos comparten sus servicios.

Además, esta malla de servicios incluye un repositorio de catálogos que contienen aquellos productos producidos por las organizaciones. Por ejemplo, imágenes médicas, señales de electrocardiogramas o expedientes clínicos electrónicos. Dichos catálogos de datos son construidos utilizando los servicios de transporte y almacenamiento de datos, los cuales permiten a las organizaciones compartir bases de datos, resultados/información y servicios.

En este contexto, una malla de servicios (Ψ) se compone como el conjunto de piezas, rompecabezas,

y meta-rompecabezas creados, como se denota a continuación:

$$\Psi = [\mathbf{F}, \mathbf{R}, \Omega, \mathbf{P}, \mathbf{RS}], \quad (4.10)$$

donde \mathbf{F} es el conjunto de fuentes de datos disponibles, \mathbf{RS} es el conjunto de destinos de datos disponibles, \mathbf{R} es el conjunto de rompecabezas creados, \mathbf{P} el conjunto de piezas y Ω el conjunto de meta-rompecabezas diseñados.

El despliegue de estos artefactos de software en la malla de servicios es posible gracias a que son paquetes de software autocontenidos y autosimilares. Tres características importantes pueden ser identificados de este método autocontenido y autosimilar:

1. El manejo de la heterogeneidad logrado al abstraer las aplicaciones como piezas de software para crear rompecabezas (estructuras de procesamiento). Esto permite que las organizaciones unan aplicaciones desarrolladas en diferentes lenguajes de programación, así como asimilar cambios el ciclo de vida de los productos digitales, simplemente reajustando las interfaces de entrada y salida de una pieza y acoplándola con otras piezas en el rompecabezas.
2. La reutilización de rompecabezas acoplándolos con otros rompecabezas, creando un meta-rompecabezas que permite la generación de flujos de datos intra e inter organizaciones.
3. La creación automática de flujos de datos a través de las piezas, rompecabezas y meta-rompecabezas es resultado de la gestión automática de los datos de entrada y salida en cada pieza.

4.3 Modelo de gestión de tareas/datos y paralelismo implícito basado en monitoreo, manejo de colas y autoescalamiento

Hasta este punto se ha presentado un modelo llamado PuzzleMesh que permite el diseño, despliegue y ejecución de estructuras de procesamiento que son manejadas como servicios agnósticos de la infraestructura. En este sentido, los componentes autocontenidos y autosimilares de PuzzleMesh, así como los modelos de comunicación y procesamiento agregados en tiempo de despliegue y ejecución, permiten agregar a las estructuras de procesamiento las características de *manejo de heterogeneidad, portabilidad y reusabilidad*, las cuales son claves cuando se construyen servicios agnósticos de la infraestructura.

4.3.1 Esquema de paralelismo implícito

En PuzzleMesh, los contenedores virtuales no solo son utilizados para agregar las características de portabilidad y reusabilidad a las aplicaciones consideradas en una estructura de procesamiento, sino también para diseñar y construir patrones de paralelismo implícitos. Estos patrones permiten a los desarrolladores crear estructuras paralelas para mejorar la eficiencia de las aplicaciones sin tener que modificar el código de estas. En PuzzleMesh, patrones paralelos tales como *gestor/trabajador* y *divide-y-vencerás* son creados clonando y encadenando la funcionalidad (F) y las aplicaciones manejadas dentro de una pieza de software. Estos patrones procesan los datos utilizando estos clones para mejorar la eficiencia de las piezas, lo cual es clave cuando se procesan grandes volúmenes de datos y en procesos de toma de decisiones.

En un patrón gestor/trabajador, una entidad llamada gestor se encarga de distribuir un conjunto

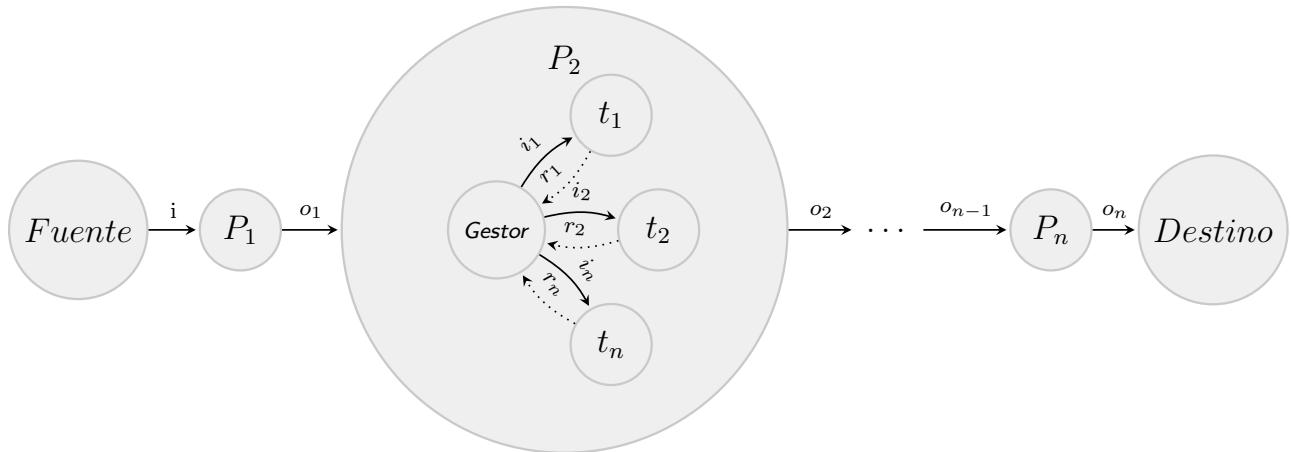


Figura 4.10: DAG de un rompecabezas que incluye un patrón paralelo.

de datos de entrada a diferentes trabajadores para que procesen el subconjunto de datos asignados. Mientras que en el patrón divide-y-vencerás, una entidad llamada *segmentador* realiza una tarea llamada *divide*, y en la cual cada archivo de entrada se segmenta en n partes que son entregados a un trabajador diferente. Los trabajadores procesan los segmentos y entregan los resultados a una entidad llamada *vencer*, que se encarga de unir los segmentos procesados en un solo resultado. En la Figura 4.10 se muestra el DAG de un rompecabezas implementando un patrón paralelo en una de sus piezas.

En PuzzleMesh los componentes para garantizar el funcionamiento de los patrones *gestor/trabajador* y *divide-y-vencerás* fueron abstraídos en tres componentes:

Gestor de entrada (GE). Implementa las funcionalidades para manejar los datos de entrada disponibles en la fuente de datos. Este componente puede ser configurado en tiempo de diseño para tomar el comportamiento de una entidad gestor o un segmentador.

Trabajadores (T). Son clones de la funcionalidad de una pieza de software, los cuales procesan los datos de entrada.

Gestor de salida (GS). Cuando se configura el patrón divide-y-vencerás, este componente toma el comportamiento de un *vencer*, en caso contrario este componente solo entrega los datos a

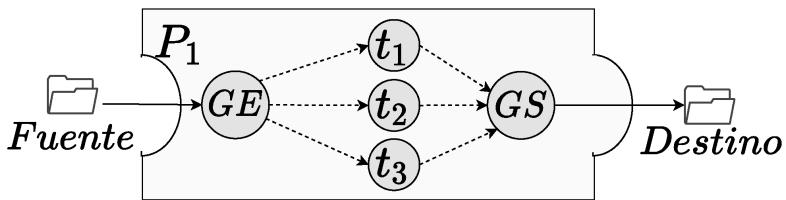


Figura 4.11: Representación conceptual de una pieza de software manejada como un patrón de paralelismo.

la siguiente pieza o a un destino de datos.

En PuzzleMesh, una pieza configurada como un patrón paralelo se denota de la siguiente manera:

$$\varphi(P) = GE(n, \text{patron}) \Rightarrow \mathbf{T} \Rightarrow GS(\text{patron}) \quad (4.11)$$

donde $\varphi(P)$ significa que la pieza P se encuentra configurada como un patrón paralelo, n es el número de trabajadores en el patrón, *patron* indica el tipo de patrón a utilizar (*mt* para gestor/trabajador y *dc* para divide-y-vencerás) y \mathbf{T} es el conjunto de trabajadores disponibles ($\mathbf{T} = \{t_1, t_2, t_3, \dots, t_n\}$). *ME* es el gestor de entrada, $t_{1,\dots,n}$ son clones de la funcionalidad $F \in P$ y *GS* es el gestor de salida.

En la Figura 4.11 se muestra una pieza diseñada como un patrón de paralelismo, el cual es ejecutado en los siguientes pasos:

1. Se crea un archivo de configuración en donde se declaran las configuraciones del patrón: número de trabajadores, tipo de trabajador y funcionalidad a clonar. En este sentido, la funcionalidad es manejada como un solo paquete de software que puede ser clonado y manejado como una *semilla*.
2. A partir de la semilla, se crean los n trabajadores y son desplegados en la infraestructura disponible.
3. Los trabajadores son asociados con los gestores de entrada y salida, las cuales están a cargo de entregar los datos a los trabajadores para que los procesen, así como de recibir los resultados

Algoritmo 1 Algoritmo de ejecución de un patrón en una pieza.

Entrada: Número de trabajadores n , datos de entrada **DE**, metadatos de la funcionalidad (F)

Salida: Estado del patrón y resultados del procesamiento **R**

```

1: semilla_trabajador  $\leftarrow$  funcionalidad  $\in P$ 
2: trabajadores  $\leftarrow$  clonar(semilla_trabajador)
3: entradas_trabajadores  $\leftarrow$  GE.distribuir(trabajadores, DE)
4: R  $\leftarrow \{\}$ 
5: monitores  $\leftarrow \{\}$ 
6: estado  $\leftarrow 0$ 
7: aux  $\leftarrow 0$ 
8: para  $n, i \in (\text{trabajadores}, \text{entradas\_trabajadores})$  hacer
9:   monitor  $\leftarrow$  lanzarTrabajador( $n, i$ )
10:  monitores.agregar(monitor)
11: fin para
12: mientras aux  $< n$  hacer
13:   para mnt  $\in$  monitores hacer
14:     estado  $\leftarrow$  mnt.estado()
15:     si estado ==FINALIZO entonces
16:       aux  $\leftarrow$  aux + 1
17:       R.add(GS.resultados(mng.trabajador))
18:     fin si
19:   fin para
20: fin mientras
21: devolver state, R

```

para entregar a la siguiente pieza o a un destino de datos.

4. El patrón es abstraído como una pieza que puede ser acoplada con otras piezas en un rompecabezas, creando estructuras de procesamiento paralelas.

Esta organización de estructuras paralelas produce piezas de software autosimilares que son manejadas en PuzzleMesh como una sola solución.

En el Algoritmo 1 se muestra el proceso para manejar un patrón en PuzzleMesh, incluyendo el despliegue de los trabajadores, la entrega de datos y la recuperación de los resultados.

Un rompecabezas (R) que incluye una pieza paralela es denotada de la siguiente manera:

$$R = [\text{fuente}_1 \rightarrow \varphi(P_1)], [\varphi(P_1) \rightarrow P_2] \quad (4.12)$$

donde P_2 es una pieza regular y es una pieza paralela construida con tres trabajadores, y que es representada de la siguiente manera:

$$\varphi(P_1) = GE(3, mw) \Rightarrow \mathbf{T} \Rightarrow GS(mw), \quad (4.13)$$

donde \mathbf{T} es el conjunto de trabajadores disponibles en el patrón ($\mathbf{T} = \{t_1, t_2, t_3\}$).

La ejecución de los clones da como resultado el procesamiento recurrente de los productos digitales, lo cual se espera mejoré el tiempo de respuesta de una pieza (dependiendo de las características de la infraestructura). Además, estas tareas de paralelismo son realizadas de forma implícita y transparente para el usuario. Por lo tanto, estas piezas paralelas realizan la misma funcionalidad que las piezas regulares e incluso que las aplicaciones dentro de la pieza.

4.3.2 Algoritmo de balanceo de carga de datos

Como se puede observar, en el patrón gestor/trabajador los datos se deben de distribuir entre diferentes trabajadores. La distribución de los datos puede ser crucial para mejorar la eficiencia un patrón paralelo, debido a que una distribución desigual puede causar dos efectos:

- El primer efecto es que se genere una sobrecarga en un trabajador cuando a este se le asigne más carga que al resto, afectando el tiempo de respuesta de la solución.
- Mientras que el segundo efecto, es que los trabajadores con menos carga estarán en estado ocioso cuando terminen de procesar su carga, la cual es menor que la del resto de trabajadores.

Algoritmo 2 Algoritmo de balanceo de carga fuera de línea.

Entrada: Lista de trabajadores (T) y conjunto de productos digitales a procesar(PD).

Salida: Lista de trabajadores con la carga de trabajo actualizada (T).

```

1:  $n \leftarrow |T|$ 
2:  $T_{cargatrabajo} = \{\}$ 
3: para todo  $p \in \text{PD}$  hacer
4:   repetir
5:      $seleccion_1 \leftarrow X \sim \mathcal{U}(n)$ 
6:      $seleccion_2 \leftarrow X \sim \mathcal{U}(n)$ 
7:     hasta que  $seleccion_1 \neq seleccion_2$ 
8:     si  $T[seleccion_1].utilizacion > T[seleccion_2].utilizacion$  entonces
9:        $trabajador \leftarrow seleccion_2$ 
10:    si no
11:       $trabajador \leftarrow seleccion_1$ 
12:    fin si
13:     $T[trabajador].item.agregar(p)$ 
14:     $T[trabajador].utilizacion \leftarrow T[trabajador].utilizacion + p.tam()$ 
15:  fin para
16: devolver  $T$ 

```

Esto produce que los recursos disponibles en la infraestructura no sean aprovechados por el patrón. Por lo tanto, en PuzzleMesh se implementa un balanceador de carga inspirado en el algoritmo *TwoChoices* [140], el cual parte de la elección de dos números pseudoaleatorios para seleccionar el trabajador al que se asignarán los datos a procesar. Este algoritmo fue implementado en dos versiones: en línea y fuera de línea. En el Algoritmo 2 se presenta el algoritmo de balanceo de carga fuera de línea. Este algoritmo es utilizado cuando se tiene que procesar un lote de datos. Similar al algoritmo TwoChoices, por cada producto digital se seleccionan dos trabajadores diferentes de forma pseudoaleatoria. Posteriormente, se evalúa la utilización, en términos de almacenamiento, de cada trabajador y se le asigna el producto digital al trabajador con la menor utilización. El algoritmo implementado en PuzzleMesh tiene la ventaja que la selección del trabajador es rápida, por lo que no impacta en el rendimiento de un patrón. Además, permite lograr una distribución cercana a la uniforme¹ con conjuntos de datos con pocos archivos.

¹Idealmente, el objetivo de un balanceador de carga es producir una distribución de datos uniforme.

Algoritmo 3 Algoritmo de balanceo de carga en línea.

Entrada: Lista de trabajadores (T) y datos a procesar(d)
Salida: Trabajador seleccionado para atender la solicitud (t)

```
1: trabs_disponibles = NULL
2: mientras trabs_disponibles == NULL hacer
3:   para  $t \in T$  hacer
4:     si t.utilizacion < t.max_capacidad entonces
5:       trabs_disponibles.agregar(t)
6:     fin si
7:   fin para
8: fin mientras
9: repetir
10:   seleccion1  $\leftarrow X \sim \mathcal{U}(\text{trabs\_disponibles}.tam())$ 
11:   seleccion2  $\leftarrow X \sim \mathcal{U}(\text{trabs\_disponibles}.tam())$ 
12:   hasta que seleccion1  $\neq$  seleccion2
13:   si trabs_disponibles[seleccion1].utilizacion > trabs_disponibles[seleccion2].utilizacion entonces
14:     trabajador  $\leftarrow \text{seleccion}_2$ 
15:   si no
16:     trabajador  $\leftarrow \text{seleccion}_1$ 
17:   fin si
18:   trabs_disponibles[trabajador].item.agregar(d)
19:   trabs_disponibles[trabajador].utilizacion  $\leftarrow \text{trabs\_disponibles[trabajador].utilizacion} + d.tam()$ 
20: devolver trabs_disponibles[trabajador]
```

En el Algoritmo 3 se muestra el algoritmo de balanceo de carga en línea implementado en PuzzleMesh.

Este algoritmo es utilizado cuando se hacen peticiones de procesamiento de datos tiempo real. Este algoritmo es similar al anterior, con la diferencia de que aquí primero se obtiene de la lista de trabajadores aquellos que se encuentren disponibles, es decir, que no estén procesando datos en ese momento. Posteriormente, se seleccionan dos trabajadores aleatoriamente y se le asignan los datos d al trabajador con la menor utilización.

Es importante considerar, que los algoritmos de balanceo de carga de PuzzleMesh consideran la heterogeneidad en el tamaño de los archivos de entrada, así como en el espacio de almacenamiento disponible en cada trabajador, lo cual no es considerado por algoritmos utilizados comúnmente en la

industria tales como Round-Robin [219].

4.3.3 Identificación y manejo de cuellos de botella en un rompecabezas

PuzzleMesh permite la implementación de patrones paralelos para mejorar la *eficiencia* de las piezas en el procesamiento de datos. Sin embargo, debido a la heterogeneidad en rendimiento y características de las aplicaciones consideradas en una estructura de procesamiento, durante tiempo de ejecución surgen cuellos de botella que impactan en el rendimiento de una estructura de procesamiento.

En un rompecabezas, un cuello de botella se puede identificar como aquella pieza de software con el rendimiento más bajo, el cual es medido como la cantidad de datos procesados por unidad de tiempo. Un cuello de botella produce los siguientes efectos en un rompecabezas:

- Limita el rendimiento del rompecabezas, debido a que la capacidad de este sistema (cantidad de datos procesados por tiempo), está limitado por el rendimiento del cuello de botella.
- Tiempos de ociosidad en las piezas posteriores al cuello de botella, debido a que este produce un efecto de acumulación, limitando el flujo de datos hacia el resto de las piezas.

En este sentido, identificar y mitigar un cuello de botella en un rompecabezas es crucial para aumentar su eficiencia [190]. Al contrario, mejorar la eficiencia de aquellas piezas de software que no son un cuello de botella produce más retrasos en un sistema (rompecabezas), debido la sobrecarga que estas piezas más rápidas produce sobre el cuello de botella.

Por lo tanto, en la presente sección se describe un modelo para la identificación y mitigación de cuellos de botella en tiempo de ejecución. Esta sección primero describe un esquema de monitoreo de las piezas en un rompecabezas, y posteriormente un esquema de manejo de cuello de botella basado en manejos de colas y técnicas de autoescalamiento.

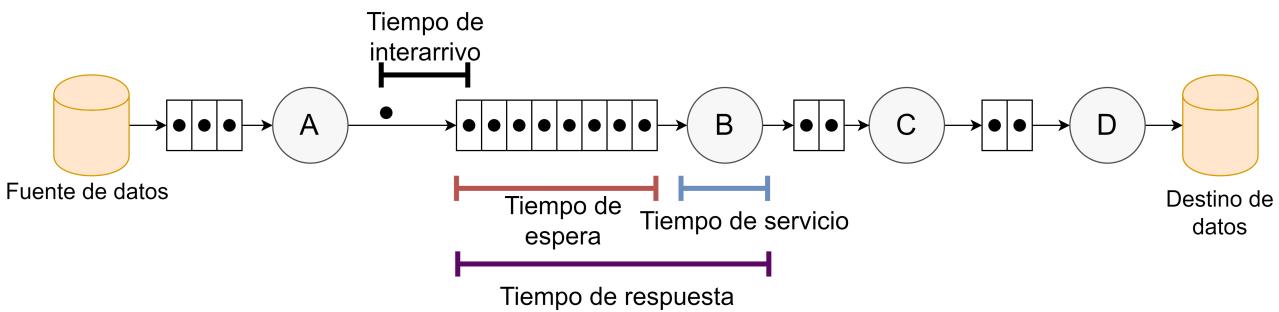


Figura 4.12: Métricas observadas y recolectadas en un rompecabezas.

4.3.3.1. Esquema de monitoreo continuo de servicios desplegados en múltiples infraestructuras

Para identificar cuellos de botella en un rompecabezas se diseñó un esquema de monitoreo que permite la recolección de diferentes métricas de desempeño de cada una de las piezas de un rompecabezas. Dichas métricas permiten observar el comportamiento de los componentes de un rompecabezas y el impacto de un cuello de botella en este. Abstrayendo un rompecabezas como una línea de producción donde cada pieza de software es una etapa de esta línea. En la Figura 4.12 diferentes etapas, podemos observar la representación de un rompecabezas como una línea de espera, en donde que cada etapa (pieza de software) tiene una cola de entrada que crece en tamaño a medida que nuevos productos digitales arriban a está, y reduce su tamaño a medida que los productos son tomados por la etapa para su procesamiento.

Las siguientes métricas se pueden medir a partir de un rompecabezas abstraído como una línea de procesamiento (ver Figura 4.12):

Tiempo de interarribo. Mide el tiempo que pasa entre el arribo de un producto digital n y el producto digital $n - 1$.

Tiempo de espera. Mide el tiempo que espera un producto digital en cola antes de ser atendido/procesado por una pieza.

Tiempo de servicio. Mide el tiempo que tarda un producto digital en ser procesado por una pieza

de software.

Tiempo de respuesta. Mide el tiempo desde que un producto digital llega a la cola de una pieza y es procesado por esta. Incluye los tiempos de lectura del producto y de escritura de los resultados.

En un cuello de botella, el tiempo de espera en cola tiende a ser mayor que el tiempo en cola del resto de piezas. Lo anterior se debe a que el cuello de botella tiene un rendimiento menor, por lo que los productos digitales de entrada deben de esperar para ser atendidos. En un cuello de botella, el tiempo de espera medio en cola es constante al inicio; sin embargo, a medida que las peticiones se acumulan, este tiempo pasa primero a ser lineal y posteriormente exponencial. En este sentido, si el crecimiento del tiempo de espera medio en cola no es contenido, se producirá una sobrecarga del sistema y éste no podrá seguir atendiendo solicitudes. Además, este tiempo de espera impacta directamente en el rendimiento del rompecabezas debido a que los tiempos de espera de todos los productos digitales a procesar se acumulan, incrementando los tiempos de respuesta.

Para obtener registros de las métricas ilustradas en la Figura 4.14, se diseñó un esquema de monitoreo continuo en el cual entidades llamadas *monitores* son conectadas a las piezas para recolectar métricas de rendimiento de las piezas. En la Figura 4.13 se muestra una representación conceptual de un rompecabezas que incluye un conjunto de monitores. En tiempo de despliegue, estos monitores son desplegados en la misma infraestructura donde se despliegan las piezas del rompecabezas. Estos monitores se conectan con las interfaces de entrada y salida de las piezas para recolectar métricas tales como el número de productos en la cola de entrada, metadatos de los productos tales como su tiempo de arribo y tamaño, tiempos de servicio y el tiempo de respuesta. Las métricas identificadas durante el monitoreo son entregadas a un *registro de eventos*, el cual es consumida por un gestor de autoescalamiento para identificar y escalar los cuellos de botella.

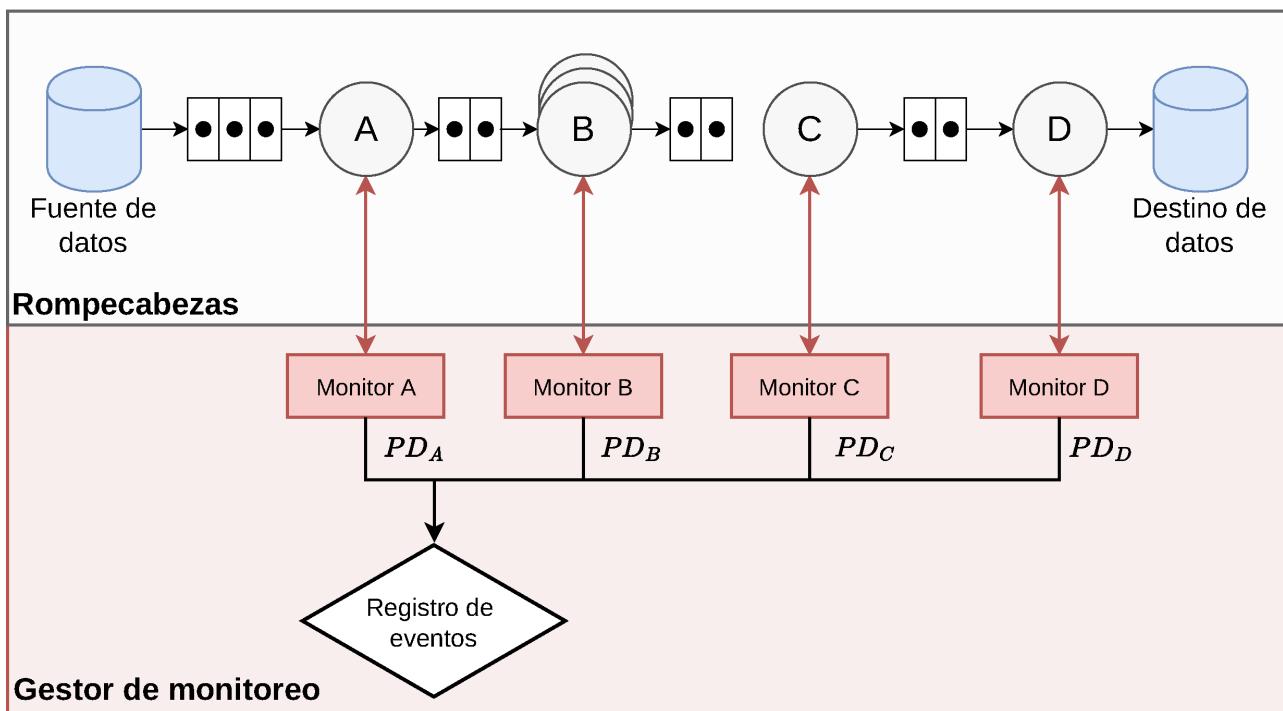


Figura 4.13: Esquema de monitoreo del rendimiento de piezas en un rompecabezas.

4.3.3.2. Identificación de cuellos de botella en un rompecabezas

Las métricas obtenidas durante el monitoreo de las etapas son entregadas a una entidad llamada *Gestor de Rompecabezas*, el cual implementa un módulo para analizar estas métricas e identificar cuellos de botella. En este sentido, primero se calcula el rendimiento o *throughput* de cada pieza de la siguiente manera:

$$th_P = \frac{|DP|}{TR}, \quad (4.14)$$

donde $|DP|$ representa la cantidad de datos procesados por una pieza hasta el momento de la última medición y TR es el tiempo de respuesta de una pieza observado para procesar estos datos. Este tiempo de respuesta incluye el tiempo de extracción, transformación y carga de cada producto digital proceso, así como su tiempo de espera en cola. Por lo tanto, el tiempo de respuesta (TR) se calcula

Algoritmo 4 Algoritmo para la identificación de un cuello de botella.

Entrada: Métricas de rendimiento de las piezas (RP)

Salida: Cuello de botella (Btl)

- 1: **para** $r \in RP$ **hacer**
 - 2: $r.th \leftarrow r.datosprocesados/r.tr$ {Se calcula el throughput de una pieza dividiendo la cantidad de datos procesados entre el tiempo de respuesta de la pieza.}
 - 3: **fin para**
 - 4: $Btl \leftarrow \text{MINTHPOS}(RP)$ {Se obtiene la pieza con el menor throughput, la cual es el cuello de botella en el puzzle.}
 - 5: **devolver** Btl
-

de la siguiente manera:

$$TR_P = \sum_{p \in \mathbf{PD}} [e(p) + t(p) + c(p) + TE(p)] \mid \mathbf{PD} = \{p_1, p_2, \dots, p_n\}, \quad (4.15)$$

donde \mathbf{PD} es el conjunto de productos digitales esperando en cola, p es un producto digital, $e(p)$ es el tiempo de extracción de un producto digital, $t(p)$ su tiempo de transformación, $c(p)$ su tiempo de carga y $TE(p)$ el tiempo de espera en cola de este producto. Un cuello de botella es aquella pieza que tiene el menor rendimiento. Este proceso de identificación de cuellos de botella se ilustra en el Algoritmo 4.

Identificado el cuello de botella en un rompecabezas, su capacidad será igual al rendimiento de este cuello de botella. La capacidad de un rompecabezas básicamente denota la cantidad máxima de datos que este puede procesar.

4.3.3.3. Esquema de mitigación de cuellos de botella

Una vez identificado el cuello de botella, el siguiente paso es mitigarlo para mejorar la eficiencia del rompecabezas y aumentar su capacidad. Por lo tanto, para mitigar un cuello de botella se utiliza un esquema de autoescalamiento basado en patrones de paralelismo que permite aumentar el número de trabajadores en la pieza identificada como cuello de botella.

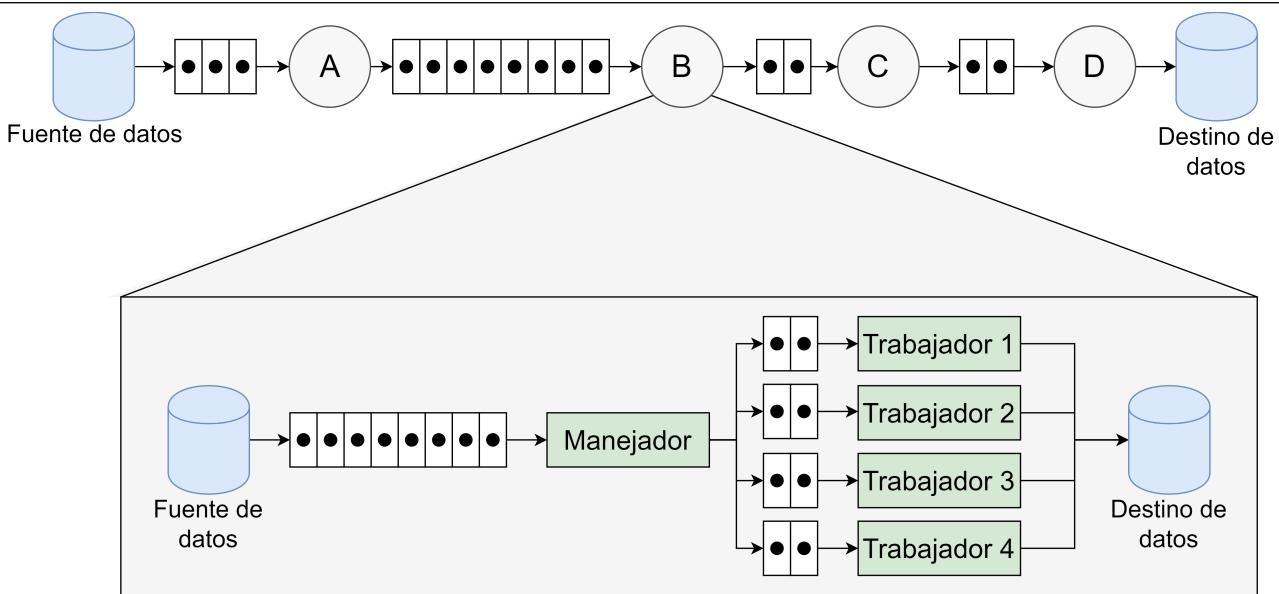
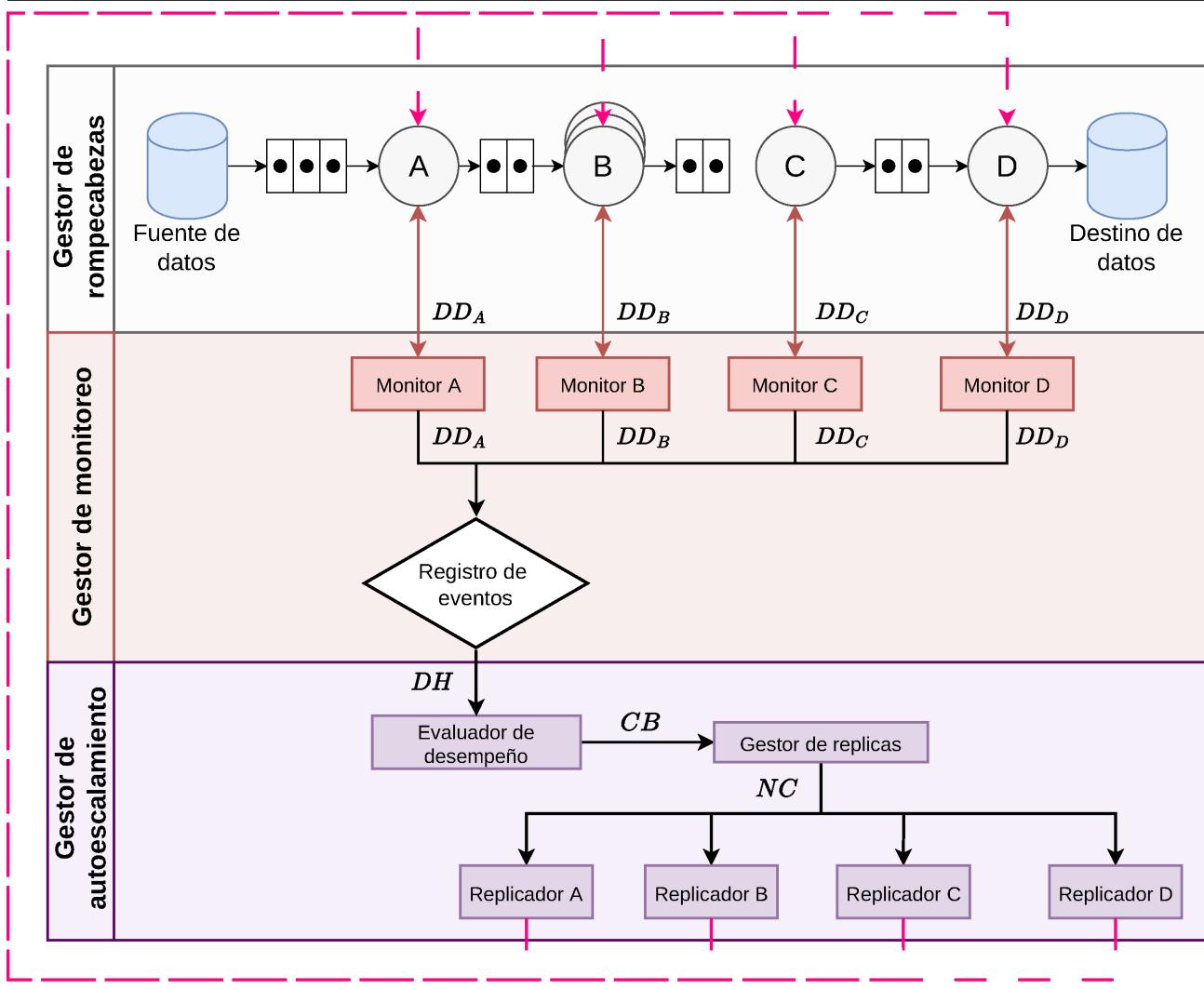


Figura 4.14: Manejo de colas utilizando patrones de paralelismo.

Los patrones de paralelismo de PuzzleMesh permiten distribuir los productos digitales entre los diferentes trabajadores de un patrón. Lo anterior reduce el tiempo en cola que un producto digital debe esperar para ser atendido. En la Figura 4.14 se muestra la representación conceptual de cómo las colas de una pieza son manejadas utilizando un patrón de paralelismo de tareas (gestor/trabajador). En este sentido, se espera que al agregar más trabajadores a una pieza los tiempos de espera en cola se reduzcan, produciendo un incremento en el desempeño de la pieza. Cabe resaltar, que este incremento está limitado por los recursos disponibles en la infraestructura donde se encuentre desplegada. Por ejemplo, comúnmente el número máximo de trabajadores, que produce un incremento en el desempeño de una pieza, es igual al número de núcleos físicos de un equipo.

Para manejar el despliegue de nuevos trabajadores en las piezas, se implementó un *gestor de autoescalamiento* conectado al gestor de monitoreo, el cual permite escalar las piezas en tiempos de ejecución para mitigar los cuellos de botella identificados durante el monitoreo. En la Figura 4.15 se muestra la representación conceptual de los componentes del gestor de autoescalamiento, el cual incluye un *evaluador de desempeño* que se encarga de la identificación del cuello de botella y elegir el número de réplicas para mitigarlo. En este sentido, el proceso para mitigar un cuello de botella



NC: Nueva configuración

CB: Cuello de botella

DH: Datos históricos

DD: Datos de desempeño

/ Emisión de la nueva configuración

Figura 4.15: Esquema de autoescalamiento de los patrones de paralelismo incluidos en las piezas.

con este gestor de autoescalamiento es el siguiente:

1. El evaluador de desempeño recupera los datos de rendimiento del registro de eventos. Con estos datos identifica el cuello de botella actual en el rompecabezas.
2. El evaluador de desempeño obtiene del registro de eventos, los datos de las dos configuraciones previas a la última configuración desplegada. En caso de que no haya los datos suficientes, se

omite este paso y el proceso continúa en el paso 4.

3. El evaluador de desempeño valora si el rendimiento de la configuración anterior es superior a las configuraciones anteriores. Dependiendo del desempeño observado, el gestor toma una de las siguientes decisiones:
 - En caso de que el rendimiento haya disminuido, se utiliza la configuración con el menor número de trabajadores para liberar el uso de recursos en la infraestructura.
 - En caso contrario, se agrega un trabajador nuevo al cuello de botella.
4. Un gestor de réplicas se encarga de entregar la configuración obtenida a entidades replicadoras asignadas a cada pieza.
5. Finalmente, los replicadores diseminan la nueva configuración a las piezas para que distribuyan datos a las nuevas instancias en los patrones.

En la Figura 4.16 se presenta un diagrama de flujo, el cual ilustra este proceso de autoescalamiento de piezas.

4.4 Arquitectura y principios de diseño

En esta Sección se describe una arquitectura que describe el método propuesto en esta tesis para desarrollar servicios agnósticos de la infraestructura para el manejo del ciclo de vida de los productos digitales. Esta arquitectura fue desarrollada como una composición de servicios basada en estructuras autocontenidoas y autosimilares. La arquitectura propuesta permite alcanzar las siguientes metas:

- Permitir el despliegue de estructuras de procesamiento (diseñadas como rompecabezas), conformadas por un conjunto de piezas autosimilares y autocontenidoas, en la infraestructura disponible.

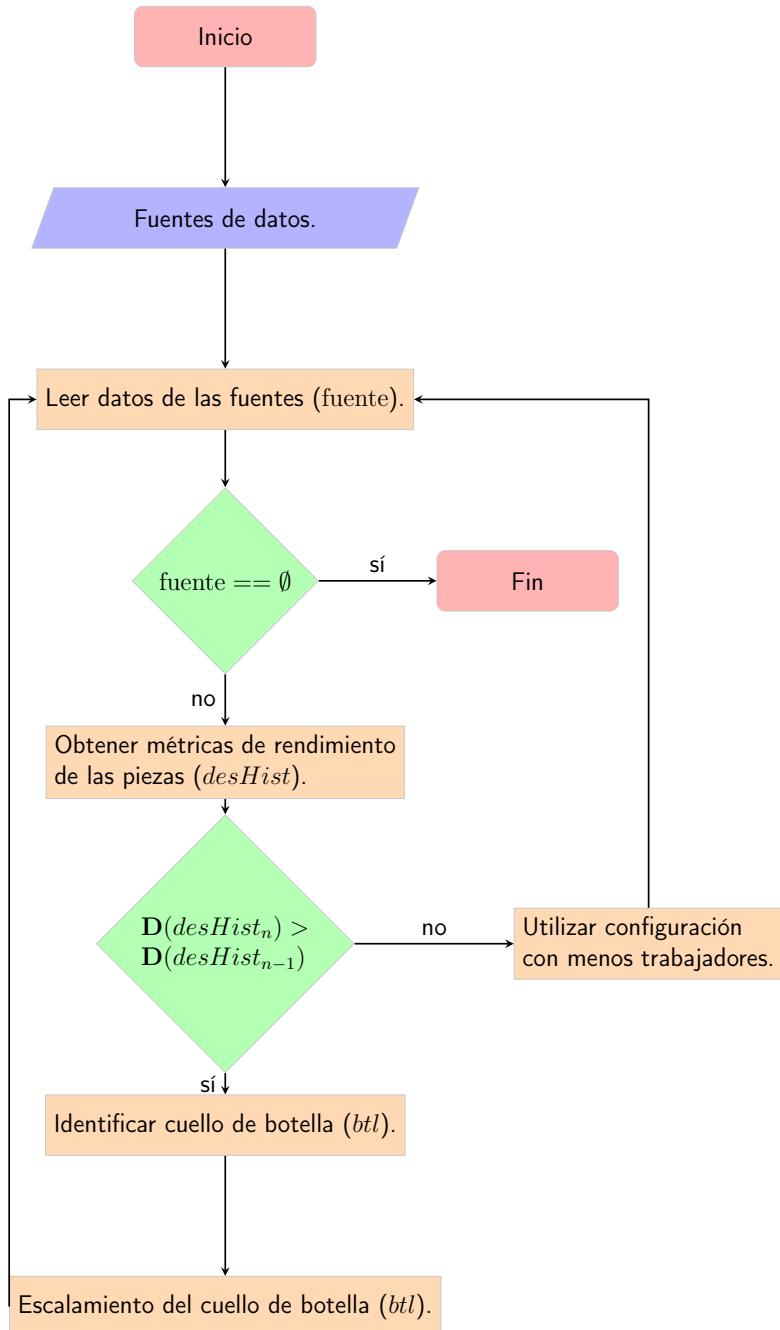


Figura 4.16: Diagrama de flujo del proceso de autoescalamiento en un rompecabezas.

- Crear un flujo de datos a través de todos los componentes de la estructura de procesamiento para el manejo del ciclo de vida de los productos digitales.
- Mitigar los cuellos de botella que puedan surgir durante tiempo de ejecución en una estructura

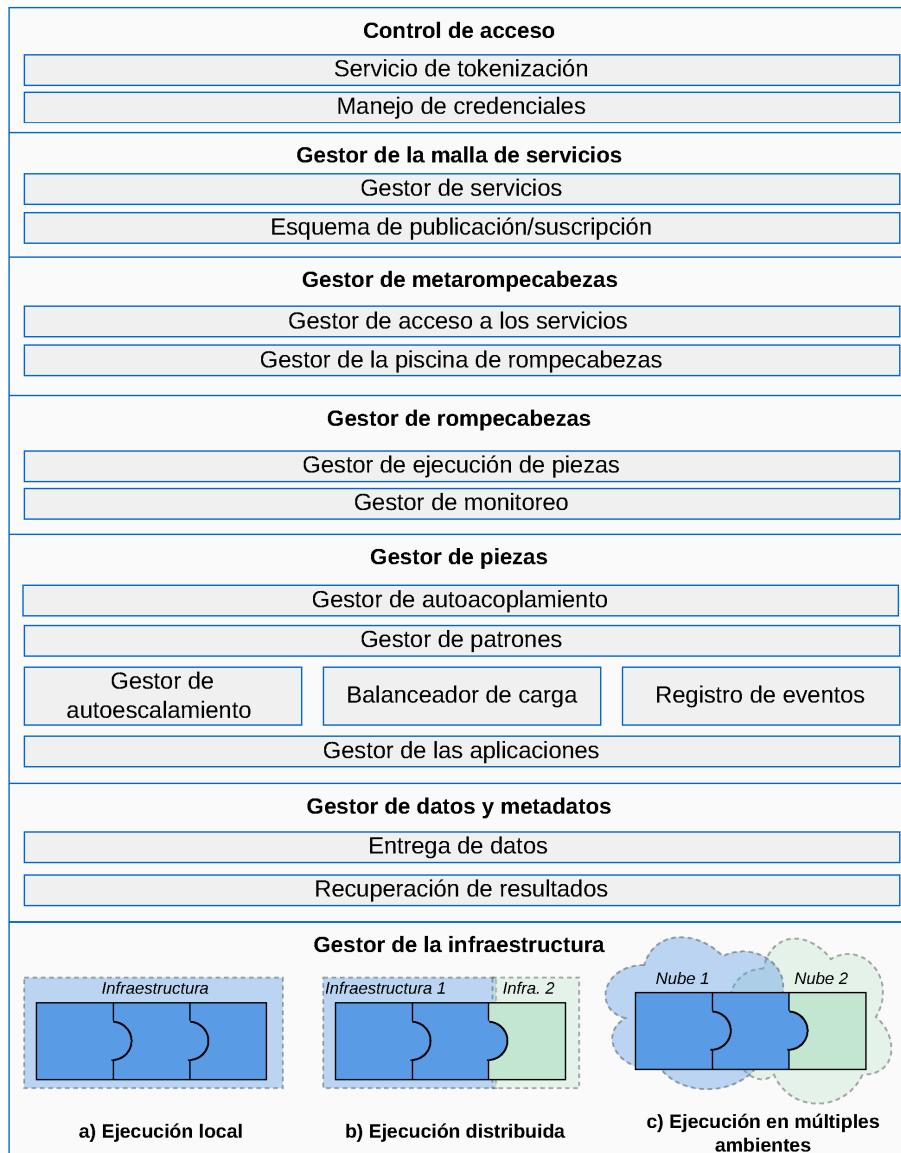


Figura 4.17: Arquitectura para el manejo de piezas, rompecabezas y meta-rompecabezas.

de procesamiento.

- Exponer las estructuras de procesamiento y sus componentes como servicios agnósticos de la infraestructura que puedan ser consumidos por los usuarios autorizados.

Es importante recalcar que esta arquitectura ha sido diseñada para manejar cada nivel de composición de un servicio agnóstico de la infraestructura, los cuales son:

- Meta-rompecabezas: flujos de datos inter e intraorganizacionales.
- Rompecabezas: estructuras de procesamiento de datos para el manejo del ciclo de vida de los productos digitales.
- Pieza de software: aplicaciones para procesar y manejar los datos en un rompecabezas.

En la Figura 4.17 se presenta la arquitectura en forma de pila para el manejo de piezas, rompecabezas y meta-rompecabezas. Como se puede observar, la pila considera las siguientes capas:

Control de acceso. Esta capa es el *front-end* que recibe las solicitudes realizadas por agentes externos (clientes y aplicaciones de terceros). Esta capa implementa un sistema de autenticación basado en tokenización, donde a cada agente externo se le asigna token que es validado cada que este hace una solicitud. Cabe resaltar que este sistema simple de tokenización puede ser remplazado por un sistema de autenticación basado en llaves pública/privada.

Tenencia múltiple. Esta capa permite manejar las solicitudes realizadas por los múltiples clientes que acceden a los servicios y que han sido validados en la capa anterior. Dado que múltiples clientes van a acceder a los mismos servicios, esta capa se encarga de aislar estas solicitudes, asignándoles espacios virtuales y privados de ejecución. Esto evita accesos no autorizados a los datos y resultados producidos por los servicios ejecutados por un cliente.

Gestor de la malla de servicios. Esta capa se encarga de manejar los servicios disponibles en la malla de servicios. Esta capa incluye un esquema de publicación/suscripción que permite a los diseñadores y desarrolladores hacer disponibles sus piezas a otros usuarios para que las puedan utilizar. En este sentido, los desarrolladores publican sus piezas en la malla de servicios para que otros usuarios se suscriban a ellas y las puedan utilizar.

Gestor de meta-rompecabezas. Esta capa se encarga de manejar el acceso a los rompecabezas creados. Incluye una componente para el control de acceso a los rompecabezas manejados como servicios. En este sentido, este servicio redirecciona las peticiones al rompecabezas que

debe de atenderla.

Gestor de rompecabezas. Esta capa se encarga de manejar la ejecución y monitoreo de las piezas que conforman un rompecabezas.

Gestor de piezas. En esta capa se manejan las piezas desplegadas. Para esto incluyen componentes para manejar el autoacoplamiento de las piezas utilizando sus interfaces de entrada y salida, manejo de patrones y el autoescalamiento de estos patrones. Además, en esta capa se realiza el manejo de la ejecución de las aplicaciones consideradas en la pieza.

Gestor de datos y metadatos. Esta capa maneja la entrega y manejo los datos requeridos por los componentes de las capas superiores. En este sentido, esta capa está implementada utilizando una red de distribución de contenidos basada en abstracciones llamadas catálogos y un esquema de publicación/suscripción [39, 82].

Gestor de la infraestructura. En esta capa se maneja el acceso a la infraestructura disponible para desplegar los servicios agnósticos de la infraestructura.

4.4.1 Diseño y manejo de rompecabezas con la arquitectura

En la Figura 4.17 se muestra la representación conceptual de una arquitectura jerárquica compuesta por un conjunto de gestores para el control de piezas, rompecabezas y meta-rompecabezas, así como de la malla de servicios y de la infraestructura.

En la etapa de diseño, los desarrolladores definen los servicios y estructuras de procesamiento seleccionando un conjunto de piezas, rompecabezas y meta-rompecabezas de la malla de servicios. Además, los desarrolladores pueden crear una nueva pieza utilizando un esquema declarativo que incluye la imagen de contenedor a utilizar y un comando de ejecución. Las interfaces de entrada y salida, así como el acoplamiento de piezas, es especificado por los programadores en un archivo de

configuración (ver Anexo B), que al ser interpretado por PuzzleMesh produce un DAG, el cual es convertido en un archivo de configuración YML que contiene la declaración de todos los servicios de un rompecabezas que posteriormente será materializado como un conjunto de contenedores virtuales.

En la fase de construcción, el gestor de la malla de servicios invoca recursivamente, siguiendo la arquitectura jerárquica, los gestores de piezas, rompecabezas y meta-rompecabezas con base en el archivo de configuración creado por el desarrollador en tiempo de diseño. En las fases de despliegue y orquestación, el gestor de servicios primero crea las imágenes de contenedor de las piezas consideradas en el archivo de configuración. Dichas piezas serán desplegadas como microservicios, y el conjunto de piezas de un rompecabezas serán manejados como una arquitectura de microservicios. En un segundo paso, los gestores leen los metadatos de cada artefacto desplegado para preparar su funcionalidad, utilizando las aplicaciones y estructuras de control (APIs, balanceadores de carga y estructuras de intercambio de datos) embebidas en las imágenes de contenedor. Además, durante esta fase, un gestor de patrones crea y acopla los componentes requeridos para implementar patrones paralelos.

En las fases de despliegue y coreografía, los gestores primero realizan recursivamente el despliegue de las instancias de contenedor utilizando las imágenes previamente creadas, y posteriormente se realiza el acoplamiento automático siguiendo la configuración de las interfaces de entrada y salida declaradas en el archivo de configuración. El resultado de esta fase es un conjunto de contenedores virtuales organizados en la forma de una estructura de procesamiento, siguiendo el diseño de piezas y rompecabezas declarado.

En las fases de ejecución y operación, la estructura de procesamiento (conformada por piezas, rompecabezas y meta-rompecabezas) primero es expuesta como un servicio, para posteriormente estar disponible para los usuarios autorizados para operarla.

El gestor de la infraestructura incluye un cliente que implementa una API para la comunicación con la infraestructura donde los servicios serán desplegados. Cabe resaltar que cada gestor establece no solo control sobre el acoplamiento de las fases en la fase de construcción, sino que además durante

el manejo de datos en tiempo de ejecución y operación.

En resumen, la construcción de servicios es realizada siguiente la arquitectura jerárquica de arriba hacia abajo, mientras que los controles de ejecución y operación son establecidos desde el fondo hacia el tope de la arquitectura. Los gestores de la malla de servicios controlan los servicios resultantes utilizando un esquema de publicación/suscripción.

4.4.2 Clientes de acceso a la arquitectura

Se diseñó un cliente que permite a los usuarios interactuar con la arquitectura previamente descrita. En este sentido, la arquitectura puede recibir cuatro tipos de *solicitudes* para manejar cada uno de estos componentes:

- Crear una nueva pieza de rompecabezas y hacerla disponible en la malla de servicios.
- Crear un rompecabezas o meta-rompecabezas a partir de las piezas disponibles en la malla de servicios.
- Desplegar un rompecabezas en la infraestructura y hacerlo disponible a los usuarios como un servicio.
- Entregar datos a los servicios.
- Recuperar los datos crudos y los resultados producidos por los servicios.

En la Figura 4.18 se muestra la pila de módulos de este cliente, el cual incluye las siguientes capas:

Front-end. Implementa una interfaz gráfica para que los usuarios puedan acceder al resto de componentes del cliente.

Gestor de credenciales. Esta capa implementa un esquema de autenticación de usuarios basado en credenciales usuario/contraseña. Además, esta capa implementa un módulo para emitir



Figura 4.18: Pila de servicios del cliente implementado para interactuar con la arquitectura.

credenciales de acceso a usuarios y organizaciones.

Gestor de servicios. Esta capa se encarga de manejar las solicitudes que hacen los usuarios a la arquitectura.

Gestor de datos. Esta capa maneja la entrega de datos a los servicios de la arquitectura para su procesamiento, así como la recuperación de los resultados producidos por la arquitectura. Por lo tanto, esta capa incluye módulos para la recuperación y preparación de datos, para manejar los requerimientos no funcionales considerados por el usuario u organización. Por ejemplo, si se incluye el requerimiento de confidencialidad, este módulo cifrará los datos entregados a la arquitectura y descifrará los resultados obtenidos de la arquitectura.

4.4.3 Implementación de un prototipo basado en la arquitectura del método propuesto

La implementación del prototipo sigue los principios de diseño de la arquitectura anteriormente descrita. Los gestores de metarompecabezas, rompecabezas y piezas se encuentran escritos en el lenguaje de programación C++17. El gestor de patrones, incluyendo el gesto de autoescalamiento y balanceo de carga, también se encuentran desarrollados en este lenguaje de programación. Los rompecabezas y las piezas que lo integran se implementaron con base en una arquitectura de

microservicios, en la cual sus componentes se comunican a través de una API Rest implementada con el lenguaje de programación Python 3.6. Los esquemas de recuperación y preparación de datos incluidos en las piezas están implementados en el lenguaje de programación C. El intercambio de datos entre componentes distribuidos se realiza mediante una red de distribución de contenidos [39] implementada en PHP7 del lado del servidor y en Java17 del lado del cliente. Esta red también incluye bases de datos PostgreSQL para la gestión de metadatos. Los contenedores virtuales utilizados para la creación y manejo de las piezas se crean utilizando la plataforma de contenedores Docker. El prototipo diseñado soporta la creación de sistemas/servicios multi-contenedores con Docker Compose así como el manejo de clústeres de contenedores con Docker Swarm y Kubernetes.

5

Evaluación experimental: metodología y resultados

En este capítulo se presenta la evaluación experimental del método propuesto llamado PuzzleMesh. Esta evaluación fue conducida en la forma de tres estudios de caso basados respectivamente el manejo de datos meteorológicos, imágenes satelitales y datos médicos.

5.1 Metodología de experimentación

Se definió una metodología de tres fases para evaluar un prototipo basado en PuzzleMesh. Cada fase corresponde a la conducción de un estudio de caso, los cuales se describen a continuación:

Fase 1 En la primera fase de la evaluación experimental fue conducido un estudio de caso basado en el diseño, despliegue y evaluación de un sistema de aplicaciones para el manejo y procesamiento de 33 años de registros meteorológicos de los 32 estados de la República Mexicana.

Fase 2 En esta fase se diseñó un sistema de aplicaciones para el manejo y procesamiento de imágenes médicas adquiridas de una estación terrestre llamada AEM-Eris [66].

Fase 3 En la tercera etapa se diseñaron dos sistemas de aplicaciones para el manejo y procesamiento de datos médicos, tales como registros de electrocardiograma e imágenes médicas.

5.2 Materiales, métricas y ambiente de pruebas

En la presente sección se presentan los conjuntos de datos e infraestructura utilizados para llevar a cabo la evaluación experimental.

5.2.1 Métricas

Las métricas seleccionadas para evaluar el rendimiento de los sistemas de aplicaciones y de los algoritmos de balanceo de carga dentro del método propuesto fueron las siguientes:

- Sistemas de aplicaciones:
 - Tiempo de servicio (TS): representa el tiempo en el que una etapa procesa un conjunto de contenidos ($\mathbf{C} = \{c_1, c_2, c_3, \dots, c_m\}$) de tamaño m . Por lo tanto, TS puede obtenerse de la siguiente manera:

$$TS = \sum_{c \in \mathbf{C}} TE(c) + TT(c) + TC(c'), \quad (5.1)$$

donde $TE(c)$ representa el tiempo de extracción o lectura del contenido c , $TT(c)$ es el tiempo requerido para transformar o procesar c con las aplicaciones consideradas en la etapa y $TC(c')$ es el tiempo requerido para cargar o escribir, a la siguiente etapa o al destino de los datos (p. ej. un espacio de almacenamiento en la nube), los resultados (c')

del procesamiento de c .

- Tiempo de respuesta (TR): representa el tiempo observado por el usuario final cuando ejecuta una estructura de procesamiento agnóstica de la infraestructura. Este tiempo se empieza a medir desde que el usuario lanza una solicitud a la estructura de procesamiento hasta que este le entrega los resultados. Por lo tanto, este tiempo se puede calcular de la siguiente manera:

$$TR = t_{fin} - t_{sol}. \quad (5.2)$$

donde t_{fin} es una estampa de tiempo que representa el tiempo en el cual el último contenido es entregado al usuario final, mientras que t_{sol} es una estampa de tiempo que se obtiene en el momento que el usuario manda una solicitud al servicio para iniciar el procesamiento de los datos. El tiempo de respuesta es una métrica útil para evaluar el rendimiento de un servicio.

■ Algoritmos de balanceo de carga:

- Carga ideal (CI): representa la carga de trabajo (medida en megabytes) que debe de ser asignada a cada trabajador en un patrón de paralelismo, para que todos los trabajadores procesen la misma cantidad de datos. CI se calcula de la siguiente manera:

$$CI = \frac{|CT|}{n}, \quad (5.3)$$

donde $|CT|$ es el tamaño total de la carga de trabajo de entrada y n es el número de trabajadores en el patrón. Esta métrica es utilizada para evaluar la efectividad de los algoritmos de balanceo de carga implementados en PuzzleMesh.

- Porcentaje de error ($\%_{error}$): mide la efectividad de un algoritmo de balanceo de carga, comparando el tamaño de los datos asignados a un trabajador (CT_x). Por lo tanto, el

Tabla 5.1: Principales características de los tres conjuntos de productos digitales utilizados para conducir cada estudio de caso.

Fase de experimentación	Tipo de producto digital	Tamaño total (GB)	Cantidad de productos	Tamaño promedio (MB)
Estudio de caso 1	Datos meteorológicos	2.1	4386	0.45
Estudio de caso 2	Imágenes satelitales Terra	393.14	1966	204.76
	Imágenes satelitales Aqua	172.15	1011	174.53
	Imágenes satelitales LandSat5	448.24	1662	275.67
	Imágenes satelitales LandSat8	38	40	911.06
Estudio de caso 3	Tomografías de hueso largo	37	70270	0.52
	Tomografías de pulmón	.58	1104	0.52
	Datos de electrocardiograma	.391	1000	0.39

porcentaje de error es calculado de la siguiente manera:

$$\%_{error} = \frac{CT_x - CI}{CI} \times 100. \quad (5.4)$$

Por ejemplo, si un trabajador tiene una carga de $CT_x = 3.47$ MB y la carga ideal es de $CI = 3.16$ MB, el porcentaje de error del algoritmo de balanceo de carga es 9.81 %.

5.2.2 Conjuntos de datos

En la Tabla 5.1 se muestra una breve descripción de las principales características de cada conjunto de datos utilizado para conducir cada estudio de caso considerado en las fases de experimentación.

En el primer estudio de caso se utilizó un conjunto de datos que contiene registros capturados por estaciones meteorológicas de la CONAGUA y adquiridos utilizando un web crawler. En total se adquirieron registros de 33 años (de 1985 a 2018) de los 32 estados de la República Mexicana y estos registros incluyen métricas como temperaturas máximas y mínimas (medidas en grados centígrados) así como datos de precipitación (medida en milímetros) [198].

En el segundo estudio de caso se consideraron imágenes satelitales adquiridas por una estación terrestre llamada AEM-Eris. El conjunto de datos contiene imágenes de plataformas como MODIS

Tabla 5.2: Características de los equipos, máquinas virtuales y contenedores utilizados para experimentación.

Fase experimental	Equipo	Proveedor	Región	Descripción	Memoria (GB)	Núcleos	Almacenamiento
Estudio de caso 1	EC2-1	Amazon EC2	us-west-2	Procesamiento	2 GB	2	Elástico
	DigitalOcean-SF	Digital Ocean	San Francisco	Procesamiento	8 GB	8	50 GB
	DigitalOcean-London	Digital Ocean	London	Procesamiento	8 GB	8	50 GB
	Disys0	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	12 GB	6	1.5 TB
Estudio de caso 2	Compute 1	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	64	12	2.7 TB
	Compute 2	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	64	12	2.7 TB
	Compute 3	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	64	12	2.7 TB
	Cloud 2-1	Amazon EC2	US-west-2	Procesamiento	32	16	600 GB
	Cloud 2-2	Amazon EC2	US-west-2	Procesamiento	32	16	600 GB
	Cloud 2-3	Amazon EC2	US-west-2	Procesamiento	32	16	600 GB
Estudio de caso 3	Compute 4	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	128	24	2.7 TB
	Compute 5	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	128	24	2.7 TB
	Compute 6	Cinvestav Tamaulipas	Tamaulipas	Almacenamiento/Procesamiento	128	24	2.7 TB
	Compute 7	-	Tamaulipas	Procesamiento	16	2	1 TB

(Aqua y Terra) [162] y LandSat 5 [222]. Además, se incluyeron datos de LandSat 8, los cuales fueron adquiridos desde el sitio *USGS Earth Explorer* [208].

Finalmente, en el tercer estudio de caso se utilizaron dos conjuntos de datos. El primero considera diferentes tomografías de pulmones, las cuales se encuentran almacenadas en formato DICOM y fueron adquiridas desde el sitio *The Cancer Imaging Archive* [47]. El segundo conjunto de datos incluye señales electrocardiograma, las cuales fueron generadas sintéticamente a partir de cuatro trazas reales.

5.2.3 Infraestructura

Para conducir la evaluación experimental se consideraron diferentes infraestructuras tanto de equipos físicos como de máquinas virtuales en las que se desplegaron los diferentes contenedores virtuales considerados en cada estudio de caso. En la Tabla 5.2 se consideran equipos físicos de un clúster de datos desplegado en el Cinvestav Tamaulipas y máquinas virtuales desplegados en diferentes proveedores en la nube (DigitalOcean y Amazon EC2).

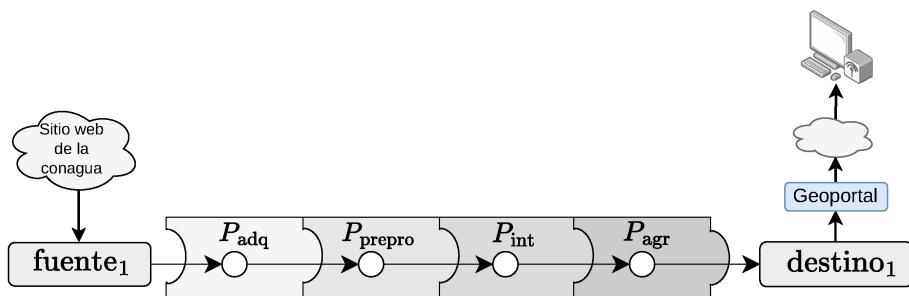


Figura 5.1: Estructura de procesamiento para el manejo de registros meteorológicos.

5.3 Estudio de caso 1: Procesamiento de registros climatológicos

En este estudio de caso, fue diseñada una estructura de procesamiento para la adquisición, preprocessamiento, integración y agrupamiento de registros meteorológicos capturados por el Servicio Meteorológico Nacional (SMN) de la CONAGUA¹. Los registros incluyen datos de los 32 estados de la república mexicana desde 1985 a 2018. En la Figura 5.1 se muestra una representación conceptual de la estructura de procesamiento, la cual incluye las siguientes piezas:

- **Adquisición (P_{adq}):** En esta pieza se incluye un crawler web para la recuperación de registros climatológicos desde el sitio web del SMN, desde donde se descargaron 2 GB de registros almacenados como archivos en formato no estructurado (archivos de texto plano). Cada archivo descargado corresponde a una antena desplegada en el territorio mexicano, y contiene información como el identificador de la antena, su ubicación geográfica (coordenadas), nombre de la estación y una tabla de 4 columnas (fecha, temperatura máxima, temperatura mínima y precipitación), y en cada fila contiene los datos disponibles para cada día desde 1985 hasta 2016.
- **Preprocesamiento (P_{prep}):** En esta pieza los registros recuperados son procesados por

¹<https://smn.conagua.gob.mx/es/>

una aplicación para transformar los archivos de texto plano en un formato estructurado (CSV). Este proceso incluye la identificación de valores como temperaturas máximas y mínimas, precipitación, fechas y ubicación geográfica. Además, se realizaron otras tareas de preprocesamiento tales como eliminación de valores anómalos y relleno de valores faltantes. Los valores anómalos fueron identificados como aquellos valores por encima o debajo del rango $[-100, 100]$ °C, mientras que los valores faltantes fueron rellenados utilizando una regresión lineal a partir de los datos históricos disponibles.

- **Integración (P_{int}):** En esta pieza los registros son consolidados en un único archivo en formato CSV. Para hacer esta consolidación, por cada estación fue calculada la media y la mediana anual de valores como la temperatura máxima y mínima, así como de precipitación.
- **Agrupamiento (P_{agr}):** En esta pieza se implementó una versión distribuida del algoritmo K-Means [198], con el objetivo de agrupar las antenas dependiendo la similitud de los registros meteorológicos de cada una. El resultado de este agrupamiento es un mapa en donde se ilustran de colores diferentes cada grupo de antenas identificado. Esta pieza puede ser ejecutada bajo demanda a través de un geoportal, permitiendo a los usuarios realizar búsquedas solo sobre una región de México en específico. La región de búsqueda es seleccionada por los usuarios dibujando un polígono sobre el mapa, el cual es enviado al servicio de agrupamiento para identificar las antenas pertenecientes a la región elegida por el usuario, y posteriormente realizar su agrupamiento.

5.3.1 Resultados experimentales

En este estudio de caso se realizaron dos experimentos:

1. En el primer experimento se evaluaron diferentes configuraciones de paralelismo creadas con Puzzlemesh. En este sentido se evaluaron tanto versiones desplegadas en un solo equipo como

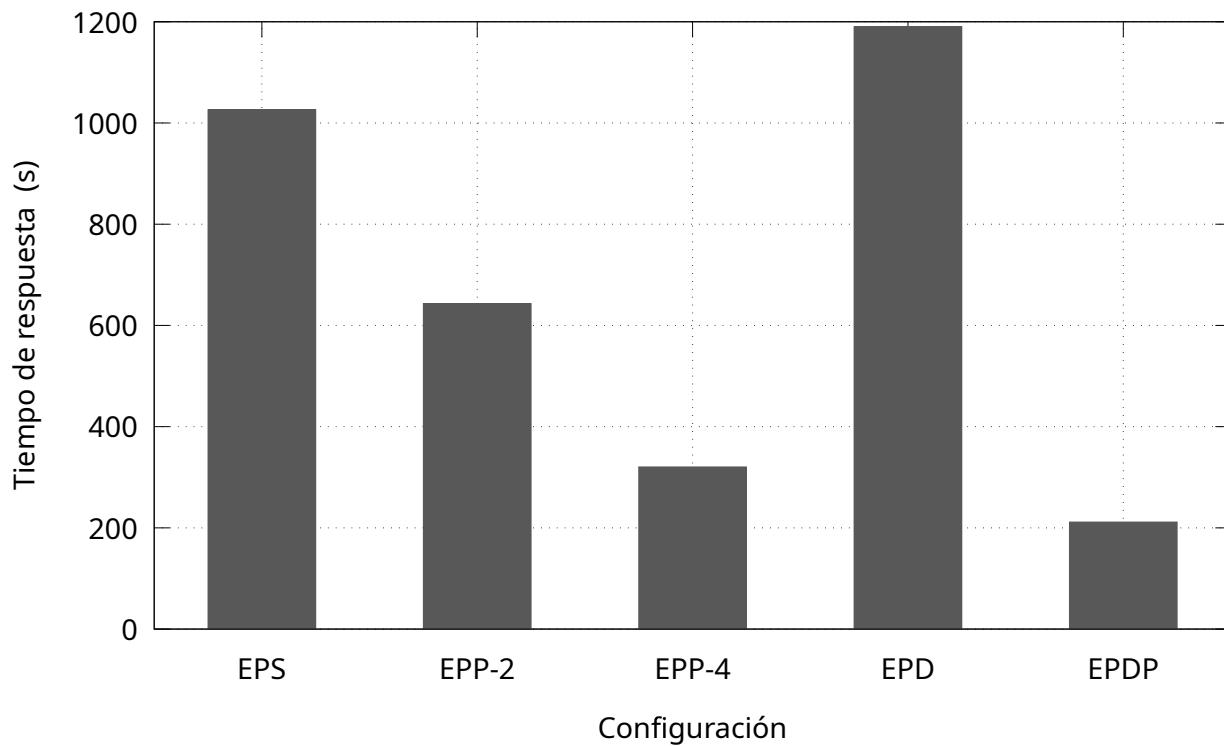


Figura 5.2: Tiempo de respuesta observado durante el procesamiento de datos meteorológicos con diferentes configuraciones creadas con PuzzleMesh.

versiones distribuidas.

2. En el segundo experimento las configuraciones que produjeron el mejor desempeño del experimento anterior fueron comparadas con herramientas del estado del arte (Makeflow [226], Parsl [16] y Pegasus [56]).

5.3.2 Experimento 1: Evaluación autocomparativa de soluciones paralelas

En este experimento se evaluaron las siguientes configuraciones creadas con PuzzleMesh:

Estructura de procesamiento secuencial (EPS). Esta estructura de procesamiento fue desplegada en un solo equipo y sin ejecutar tareas en paralelo. Esta configuración básicamente

es la solución secuencial mostrada en la Figura 5.1.

Estructura de procesamiento paralela- N (EPP- N). En esta configuración la estructura de procesamiento fue desplegada en un solo equipo y utilizando N trabajadores en paralelo para el procesamiento de datos. Esta versión fue evaluada utilizando 2 y 4 trabajadores paralelos.

Estructura de procesamiento distribuida (EPD). En esta configuración cada pieza de la estructura de procesamiento fue desplegada en diferentes equipos sin realizar tareas en paralelo: la pieza de adquisición fue desplegada en el equipo *DigitalOcean-SF*, la pieza de preprocessamiento fue desplegada en el equipo *DigitalOcean-London*, la pieza de integración en el equipo *EC2-1* y la pieza de agrupamiento en el equipo *Disys0*. Cabe resaltar que esta versión distribuida incluye el intercambio de datos a través de la red. Lo anterior fue realizado utilizando una red de distribución de contenidos llamada SkyCDS [82], la cual se incluye en PuzzleMesh como una pieza extra para el manejo y almacenamiento de datos.

Estructura de procesamiento distribuida y paralela (EPDP). En esta configuración se evaluaron las piezas distribuidas de la configuración EPD, ejecutando cada pieza de forma paralela con 8 trabajadores.

En la Figura 5.2 se muestran los resultados observados de procesar los 2 GB de datos meteorológicos a través las piezas de la estructura de procesamiento diseñada en este estudio de caso. El eje vertical de la Figura 5.2 muestra el tiempo de respuesta en segundos observado con cada configuración evaluada (eje horizontal). Como se esperaba, las configuraciones paralelas de la estructura de procesamiento reducen el tiempo de respuesta en comparación con las versiones secuenciales. Por ejemplo, el tiempo de respuesta para la configuración EPS es de, 1026.35 segundos, mientras que para la configuración EPP-4 es de 320.36 segundos. Por lo tanto, se observó una ganancia en el tiempo de respuesta de 68.7 % de la versión paralela (EPP-4) en comparación con la versión secuencial (EPS).

De forma similar, en las versiones distribuidas de la estructura de procesamiento se observó una

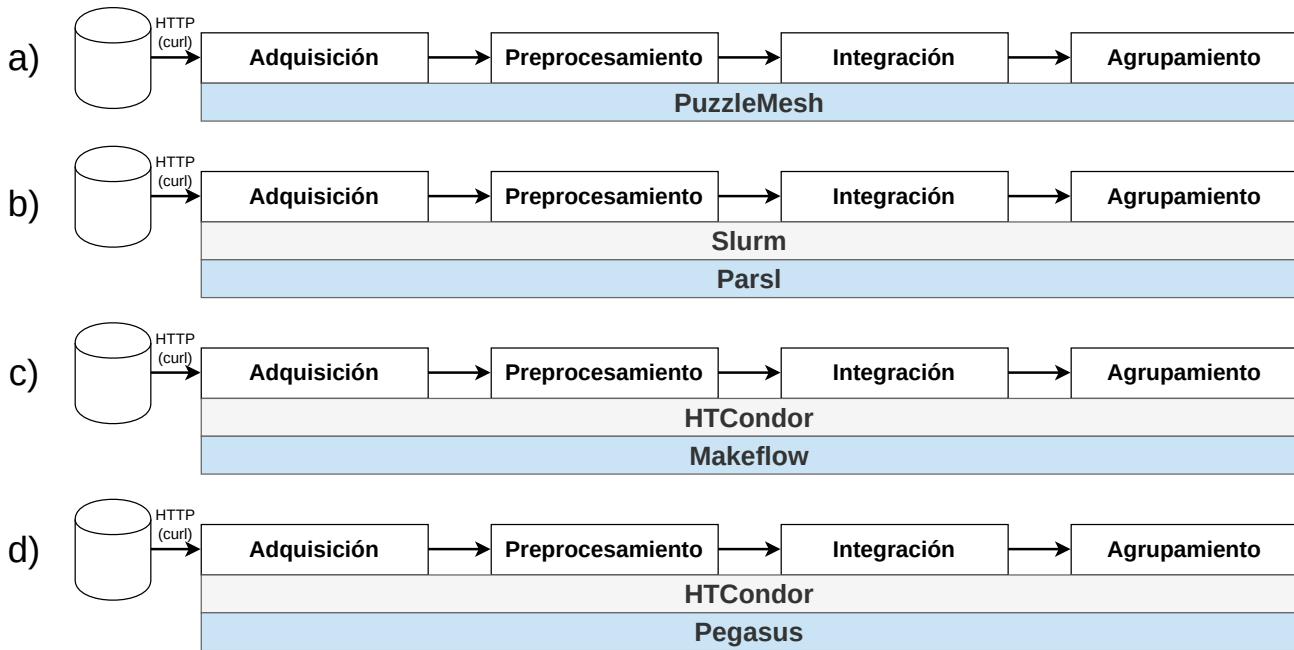


Figura 5.3: Configuración general de cada herramienta utilizada para conducir el segundo experimento de este estudio de caso.

reducción del tiempo de respuesta, comparando las configuraciones EPD (secuencial) y EPDP (paralela). La configuración EPD proceso los 2 GB de datos a través de cada pieza en un tiempo de respuesta de 1190.17 segundos, mientras que la configuración EPDP realizó el mismo proceso en 211.45 segundos. Esto significa una mejora del 82.23 % comparando la configuración paralela con la secuencial.

5.3.3 Experimento 2: Comparación de PuzzleMesh con herramientas del estado del arte

En el segundo experimento la configuración EPDP de PuzzleMesh se comparó con tres herramientas del estado del arte: *Parsl*, *Makeflow* y *Pegasus*. Por lo tanto, cada solución fue configurada siguiendo las especificaciones de las herramientas de manejo de datos e infraestructura (p. ej. HTCondor [69] y Slurm) requeridas. Para realizar este experimento las soluciones se desplegaron en un solo equipo

(EC2-1). En la Figura 5.3 se muestra la configuración de las herramientas consideradas en esta comparación. Las soluciones del estado del arte evaluadas en este experimento fueron las siguientes:

Parsl (NC=8). En esta solución, Parsl fue configurado para utilizar 8 núcleos de procesamiento en paralelo. Además, se configuró Slurm para manejar la distribución de tareas en los recursos disponibles, así como Globus para el manejo de la transferencia de datos.

Pegasus. Solución de Pegasus sin utilizar herramientas de terceros. Esta solución no contempla el procesamiento de datos en paralelo.

Pegasus + HTCondor (NC=8). Solución de Pegasus que incluye HTCondor para el manejo de la infraestructura y el procesamiento de datos en paralelo utilizando 8 núcleos. Además, en esta solución se utilizó SCP para el intercambio de datos.

Makeflow + HTCondor (MH=8). Solución de combinada con HTCondor configurada para desplegar máximo 8 hilos y utilizar SCP para el intercambio de datos.

Makeflow + HTCondor (MH=8, NM=8). Solución similar a la anterior configurando el número máximo de núcleos de Makeflow en 8.

En la Figura 5.4 se muestra el tiempo de respuesta (eje vertical) producido por cada solución evaluada (eje horizontal), incluyendo PuzzleMesh utilizando 8 trabajadores distribuidos (EPDP). Como se puede observar, el desempeño de PuzzleMesh es competitivo en comparación con el desempeño del resto de soluciones evaluadas. PuzzleMesh produjo un porcentaje de ganancia en el tiempo de respuesta de 8.01 %, 23.06 % y 2.84 % en comparación con Parsl (MH=8), Makeflow + HTCondor (MH=8) y Makeflow + HTCondor (MH=8, NM=8), respectivamente.

Cabe mencionar que estos resultados solo muestran el desempeño de cada solución bajo las condiciones configuradas, las cuales fueron elegidas desde el punto de vista del usuario final del estudio de caso. Los resultados pueden cambiar cuando se cambie la configuración o características de la infraestructura utilizada, por ejemplo, cambiando los ejecutores o gestores de la infraestructura

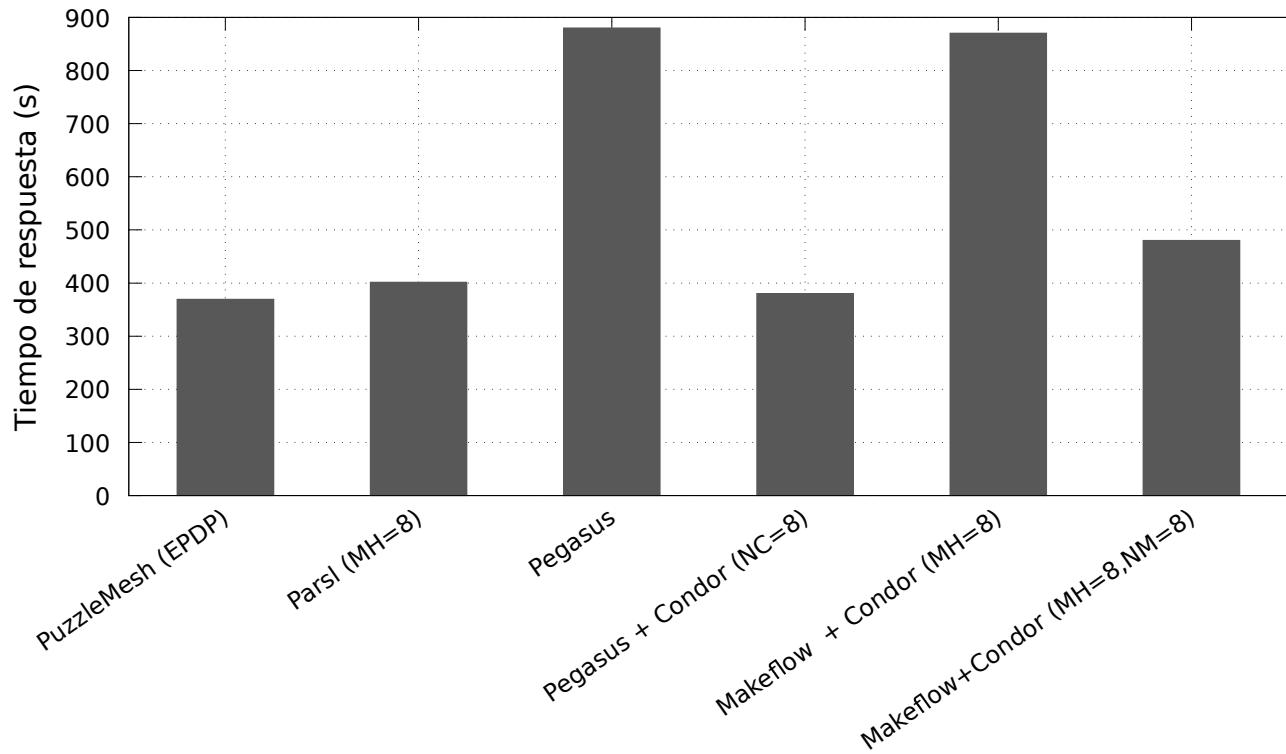


Figura 5.4: Tiempo de respuesta producido por las diferentes herramientas evaluadas. MH: Número máximo de hilos configurado. NC: Número de núcleos configurado en HTCondor. NM: Número máximo de núcleos configurado en Makeflow.

en Parsl o Makeflow.

5.4 Estudio de caso 2: Manejo de imágenes satelitales

En esta sección se presentan los resultados de un estudio de caso diseñado para el procesamiento y manejo de imágenes satelitales capturadas por la estación terrestre AEM-ERIS [66].

En la Figura 5.5 se muestra la representación conceptual de la estructura de procesamiento diseñada para conducir este estudio de caso. Esta estructura contempla tres rompecabezas para el procesamiento de datos: Adquisición (ver piezas *Lectura*, *Enriquecimiento* e *Indexamiento* en la Figura 5.5), Manufactura (ver piezas *Correcciones radiométricas*, *Correcciones atmosféricas* y

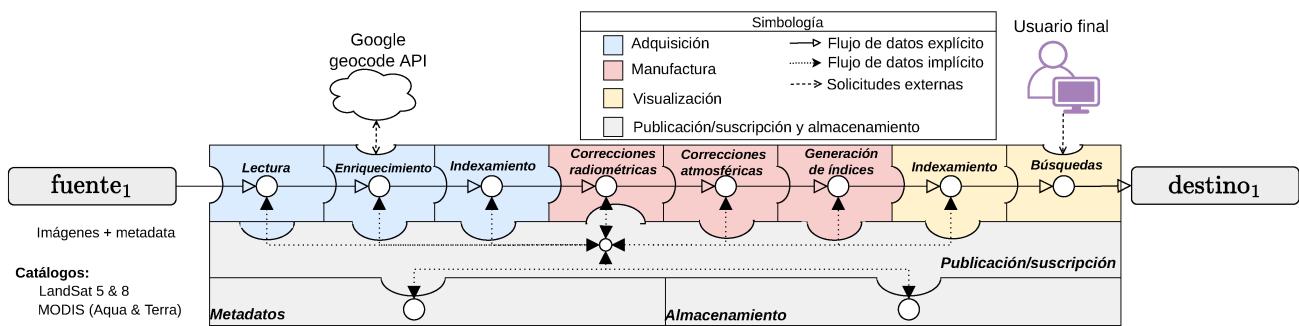


Figura 5.5: Representación conceptual de la estructura de procesamiento desarrollada para conducir este estudio de caso.

Generación de índices en la Figura 5.5) y Visualización (ver piezas Indexamiento y Búsquedas en la Figura 5.5). Además, se incluye otro rompecabezas para la gestión de datos y metadatos, el cual incluye tres piezas transversales para la Publicación/Suscripción de productos, su almacenamiento y manejo de los metadatos. En PuzzleMesh estas piezas fueron organizadas en la forma de cinco rompecabezas que están integrados como un solo meta-rompecabezas para el manejo de las imágenes satelitales.

Este meta-rompecabezas inicia con el rompecabezas de adquisición de datos, el cual incluye piezas para leer imágenes en crudo desde un servidor. Una segunda pieza se encarga de leer los metadatos de las imágenes para posteriormente enriquecerlos utilizando fuentes externas (p. ej. Google Geocoding API), para obtener a partir de las coordenadas de la imagen su dirección (ciudad, municipio, estado y país), en caso de estar disponible. Finalmente, los metadatos son indexados en una base de datos implementada en PostgreSQL.

En el rompecabezas de Manufactura (desplegado en la niebla) se procesan las imágenes satelitales para producir diferentes productos derivados a través de tres piezas. En la primera pieza se incluye un algoritmo para realizar la corrección radiométrica de cada banda en imágenes (16/32 bandas). En la segunda pieza se realiza la corrección atmosférica de cada banda. La última pieza ejecuta una aplicación para la construcción de productos derivados a partir de las imágenes corregidas. Esta pieza produce 7 tipos de productos derivados basados en diferentes índices: Índice de Vegetación

de Diferencia Normalizada, Índice de Vegetación Mejorado, Índice de Vegetación Ajustado al Suelo, Índice de Vegetación Ajustado al Suelo Modificado, Índice de Humedad de Diferencia Normalizada, Índice de Calcinación Normalizada e Índice de Diferencia Normalizada Nieve [212].

A su vez, el rompecabezas de Visualización es un servicio de geoportal que incluye el indexamiento de los productos derivados y una interfaz gráfica para su búsqueda.

En los tres rompecabezas se entregan y recuperan productos digitales, sus productos derivados y sus metadatos utilizando los rompecabezas de almacenamiento, publicación/suscripción y almacenamiento.

En este estudio de caso se realizaron dos experimentos que se describen a continuación:

Experimento 1. En este experimento se evaluaron diferentes balanceadores de carga con el propósito de identificar el algoritmo dentro de PuzzleMesh que proporcione un mejor rendimiento en términos de almacenamiento.

Experimento 2. En este experimento se evaluó el rendimiento del meta-rompecabezas implementado en PuzzleMesh, y se comparó con diferentes herramientas del estado del arte: Pegasus[56], Makeflow[226], DagOnStar[180] y Jenkins[26].

5.4.1 Experimento 1: Evaluación de balanceadores de carga incluidos en patrones paralelos

En el primer experimento de este estudio de caso, fue realizada una comparación directa de tres algoritmos de balanceo de carga implementados en PuzzleMesh: TwoChoices [140], Random [4] y RoundRobin [140]. Los tres algoritmos fueron evaluados distribuyendo una carga de datos a 12 trabajadores en un patrón paralelo. La carga de trabajo es el total de datos que un trabajador tiene que procesar. Para realizar este experimento se utilizó un conjunto de datos de 68 GB de tamaño (575

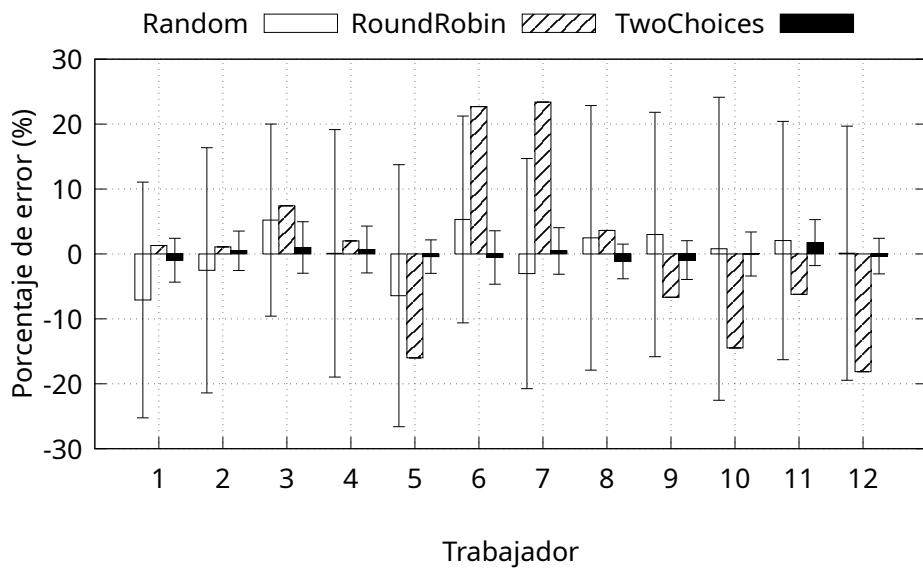


Figura 5.6: Porcentaje de error en el balanceo de carga observado con tres diferentes平衡eadores de carga.

imágenes satelitales de tamaño heterogéneo), el cual fue distribuido a los 12 trabajadores paralelos. Por lo tanto, la carga de trabajo ideal (CI) es igual a 5.66 GB por trabajador.

En la Figura 5.6 se muestra en el eje vertical el porcentaje de error observado en cada trabajador (eje horizontal) utilizando los tres平衡eadores de carga considerados en esta evaluación. Como se puede observar, el algoritmo TwoChoices fue el que produjo una carga de trabajo cercana a la ideal, con un porcentaje de error menor que 1.7 %. Mientras que Round-Robin, comúnmente utilizado en la literatura, produjo un porcentaje de error de 23.33 %. Lo anterior se debe a que este algoritmo no considera la heterogeneidad de los archivos en la carga de trabajo de entrada ni la heterogeneidad del espacio de almacenamiento de los trabajadores o nodos de almacenamiento. Por lo anterior, para conducir la siguiente etapa de esta evaluación se utilizará el algoritmo de TwoChoices.

5.4.2 Experimento 2: comparación directa con herramientas del estado del arte

En el segundo experimento de este estudio de caso, el desempeño de PuzzleMesh fue comparado con diferentes soluciones del estado del arte. Para este fin, y similar al estudio de caso anterior, se desarrolló y configuro una estructura de procesamiento, incluyendo las piezas de adquisición, manufactura y visualización, utilizando las siguientes herramientas:

DagOnStar es un motor de flujos de trabajo que produce paralelismo de tareas implícito [180].

Parsl es una herramienta que produce flujos de datos paralelos [16]. Esta solución fue configurada con Slurm para el manejo de los recursos [43].

Makeflow es un motor de flujos de trabajo que utiliza HTCondor para el manejo de los recursos de infraestructura [226].

Jenkins es una plataforma que permite crear tuberías de integración y desarrollo continuos [26]. Esta solución incluye paralelismo basado en hilos.

En este experimento se realizaron dos pruebas: la primera ejecutando los servicios en un clúster local ubicado en el Cinvestav Tamaulipas, y el segundo utilizando recursos virtualizados en la nube pública con el proveedor de servicios Amazon EC2. Para permitir el movimiento de datos entre etapas en DagOnStar, Makeflow, Jenkins y Parsl, se implementó un servicio de SCP. Mientras que en PuzzleMesh este movimiento de datos es realizado por la pieza de Almacenamiento, la cual incluye una red de distribución de contenidos [82]. En DagOnStar, Makeflow, Jenkins y Parsl se utilizó un modelo de paralelismo basado en hilos, y el número de hilos fue variado para mostrar el comportamiento de cada solución al desplegar 1, 2, 4, 8 y 16 hilos paralelos.

Cada servicio fue instalado en la misma infraestructura, pero no ejecutado al mismo tiempo. En este sentido, PuzzleMesh y las herramientas del estado del arte fueron desplegadas y ejecutadas

en el mismo ambiente, procesando los mismos datos de entrada, realizando las mismas pruebas y capturando las métricas de rendimiento en cada ejecución. La métrica evaluada en este experimento fue el tiempo de respuesta, el cual representa la experiencia de los usuarios finales cuando utilizan cada solución.

En la Figura 5.7(a) se muestran los resultados obtenidos de desplegar y ejecutar cada solución en recursos locales, mientras que la Figura 5.7(b) muestra los resultados observados en la nube. Ambas Figuras muestran en el eje vertical el tiempo de respuesta en minutos observado con cada solución utilizando un diferente número de trabajadores o hilos paralelos (eje horizontal).

La Figura 5.7(a) muestra que PuzzleMesh procesó los datos de entrada en 45.54 minutos utilizando 16 trabajadores. Mientras que DagOnStar, Parsl, Makeflow y Jenkins realizaron el mismo proceso en 54.75, 57.94, 60.94 y 66.26, respectivamente. Esto significa una mejora en el tiempo de respuesta de 16.81 %, 21.39 %, 25.26 % y 31.26 % de PuzzleMesh en comparación con DagOnStar, Parsl, Makeflow y Jenkins, respectivamente.

Un comportamiento similar al anterior fue observado cuando se ejecutó cada solución en la nube. En la Figura 5.7(b) se puede observar que PuzzleMesh procesó la carga de datos en 20.54 minutos utilizando 16 trabajadores en los patrones, mientras que DagOnStar, Parsl, Makeflow, y Jenkins realizaron la misma operación con el mismo número de trabajadores en 30.75, 29.94, 33.94 y 35.26 minutos, respectivamente. Lo anterior significa una mejora en el tiempo de respuesta de 33.18 %, 31.38 %, 39.47 % y 41.73 % de PuzzleMesh en comparación con DagOnStar, Parsl, Makeflow y Jenkins, respectivamente.

Como se puede observar, las soluciones desplegadas en la nube reducen el tiempo de respuesta en comparación con las versiones desplegadas en el clúster local. También se puede observar que el porcentaje de ganancia de PuzzleMesh en comparación con el resto de las soluciones aumenta. Lo anterior, se puede explicar debido a los cambios en la infraestructura. Los equipos en la nube tienen una mayor cantidad de núcleos, lo cual permite desplegar un mayor número de trabajadores

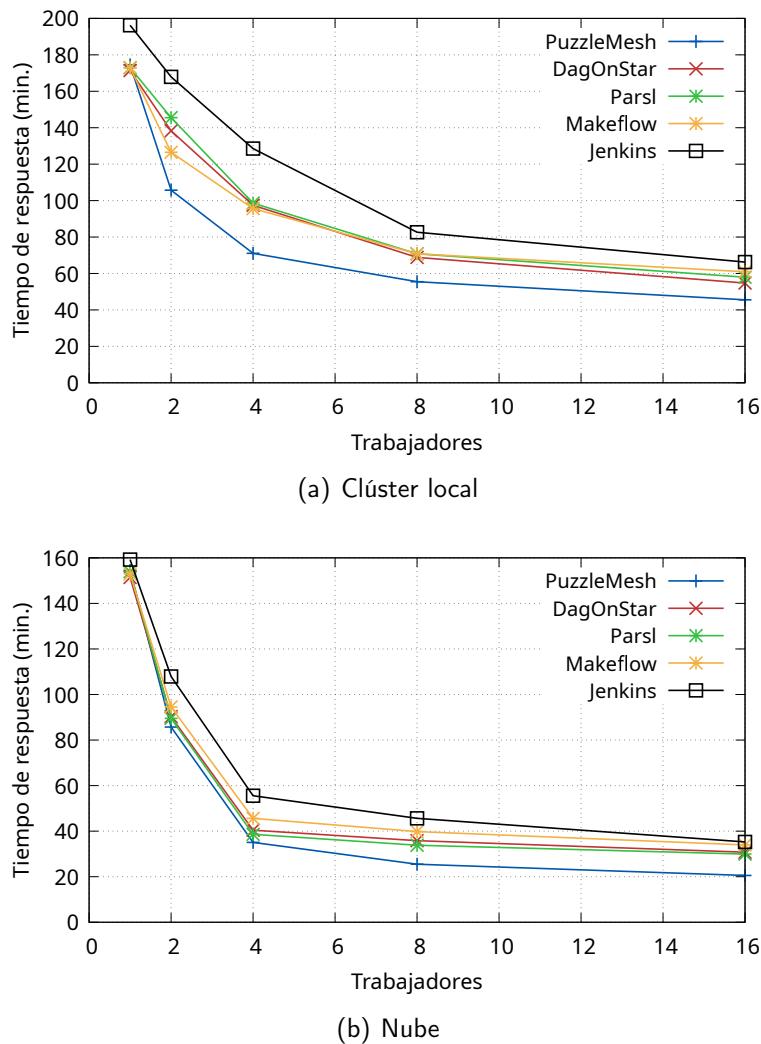


Figura 5.7: Tiempo de respuesta observado durante el procesamiento de imágenes satelitales en *a)* un clúster local y *b)* la nube.

en paralelo sin afectar el rendimiento de las soluciones.

Cabe resaltar que los resultados denotan que el desempeño de PuzzleMesh es competitivo con diferentes herramientas del estado del arte. Lo anterior, bajo las condiciones y especificaciones del estudio de caso conducido. El desempeño de estas soluciones puede variar con diferentes configuraciones de la infraestructura.

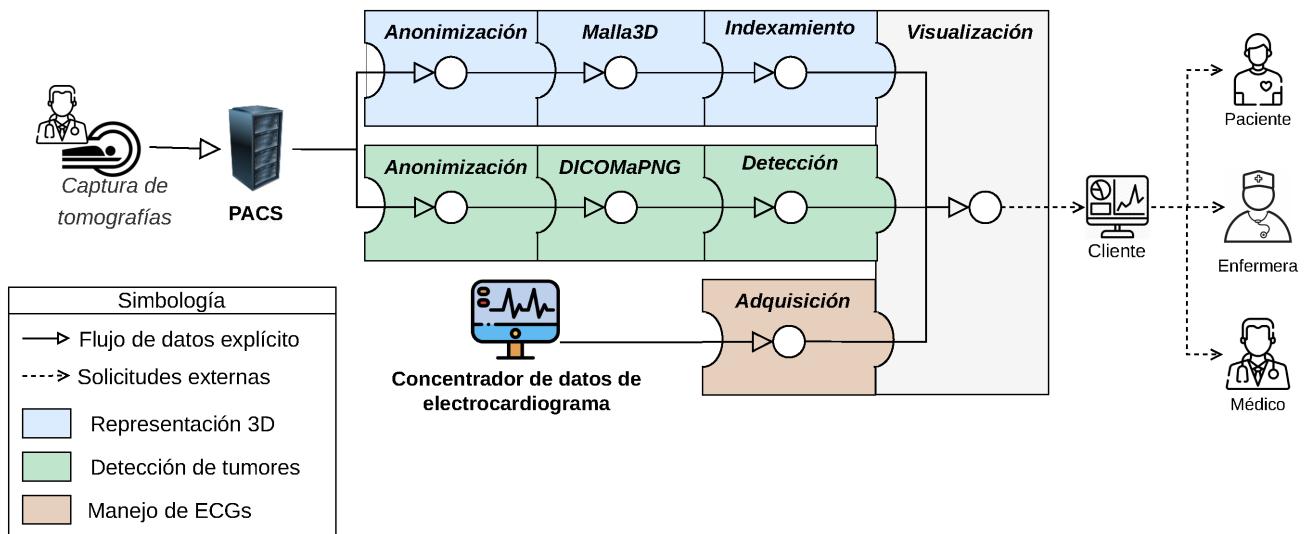


Figura 5.8: Meta-rompecabezas diseñado para el manejo y procesamiento de datos médicos.

5.5 Estudio de caso 3: manejo y procesamiento de datos médicos

En este estudio de caso se construyó un meta-rompecabezas compuesto por tres rompecabezas para el manejo de imágenes de tomografía y datos de electrocardiograma (ECG). En la Figura 5.8 se muestra el diseño de este meta-rompecabezas construido con PuzzleMesh que incluye los siguientes rompecabezas:

- El primer rompecabezas está enfocado en la reconstrucción 3D de imágenes de tomografía de huesos. Este servicio incluye tres tipos de piezas:

Anonimización. Esta pieza incluye una aplicación que elimina la información personal de las imágenes y sus metadatos para proteger la privacidad de los pacientes.

Malla3D. Genera una representación 3D a partir de un conjunto de imágenes de tomografía.

Indexamiento. Registra los productos obtenidos en las dos piezas anteriores, así como las imágenes en crudo en una base de datos implementada en MongoDB.

- El segundo rompecabezas se enfoca en la detección de tumores en tomografías de pulmón. En este caso, este rompecabezas incluye las siguientes piezas:

Anonimización. Es la misma pieza utilizada en el rompecabezas anterior para remover datos personales de las imágenes.

Conversión a PNG. Transforma las tomografías capturadas en formato DICOM a un formato PNG.

Detección. Realiza la detección de tumores en pulmones utilizando una aplicación de aprendizaje máquina, la cual implementa una red neuronal convolucional llamada faster R-CNN [79].

- El tercer rompecabezas fue diseñado para adquirir datos de ECG de un conjunto de sensores desplegados en el borde. Por lo tanto, este rompecabezas incluye una sola pieza para recibir los datos y entregarlos a la nube para su almacenamiento.

Además, la interfaz de salida de la última pieza en cada rompecabezas fue configurada para entregar los datos a la nube para su almacenamiento. En este caso, estas interfaces incluyen un esquema de preparación de datos con las siguientes aplicaciones para el manejo de requerimientos no funcionales en los datos:

LZ4. Para agregar el requerimiento de costo-eficiencia a los datos, reduciendo el volumen de datos a transportar y almacenar en la nube.

IDA. Esta aplicación divide los datos en n segmentos con redundancia llamados dispersos. Cada disperso es almacenado en una infraestructura diferente, y m segmentos son suficientes para recuperar los datos originales. Por lo tanto, este algoritmo permite tolerar $n - m$ fallas en la infraestructura de almacenamiento.

AES. Cifrado simétrico que agrega la propiedad de confidencialidad a los datos [52].

CP-ABE. Agrega el requerimiento de control de acceso a los datos creando un sobre digital que solo puede ser abierto por los usuarios que tengan un conjunto de atributos válidos [114].

5.5.1 Diseño de experimentación y resultados experimentales

En este estudio de caso se condujeron ocho experimentos para evaluar diferentes características de PuzzleMesh. Estos experimentos son los siguientes:

Experimento 1. En este experimento se midieron los costos de despliegue de una solución de PuzzleMesh. Este experimento también se utilizó para demostrar la capacidad de PuzzleMesh de desplegar soluciones en múltiples infraestructuras.

Experimento 2. Procesamiento paralelo de imágenes médicas utilizando PuzzleMesh. En este experimento se evaluó la eficiencia de diferentes configuraciones paralelas creadas con PuzzleMesh.

Experimento 3. Análisis estadístico de las diferentes configuraciones paralelas evaluadas en el Experimento 2, para demostrar que las diferencias en rendimiento producidas por cada configuración paralela son estadísticamente significativas.

Experimento 4. Análisis de los costos de preparación de datos para agregar los siguientes requerimientos no funcionales a los datos: costo-eficiencia, confiabilidad, confidencialidad y control de acceso.

Experimento 5. Comparación del rendimiento de PuzzleMesh con una solución ad-hoc para demostrar la eficiencia de PuzzleMesh para crear diferentes soluciones para el manejo del ciclo de vida de las imágenes médicas.

Experimento 6. Comparación directa de PuzzleMesh con herramientas del estado del arte.

Experimento 7. Evaluación del modelo de mitigación de cuellos de botella incluido en PuzzleMesh.

Tabla 5.3: Resumen de los experimentos diseñados para cumplimentar los requerimientos no funcionales de portabilidad, manejo de la heterogeneidad, reusabilidad, eficiencia y confiabilidad.

Experimento	Objetivo	Portabilidad	Eficiencia	Característica evaluada	Reusabilidad	Confiabilidad
				Manejo de la heterogeneidad		
1	Evaluación de costos de despliegue.	*	-	-	-	-
2	Evaluación de patrones paralelos.	-	*	*	-	-
3	Ánálisis estadístico.	-	*	-	-	-
4	Ánálisis de los costos de preparación de datos.	-	*	*	-	*
5	Comparación con una solución ad-hoc.	-	*	-	-	-
6	Comparación con herramientas del estado del arte.	-	*	-	-	-
7	Mitigación de cuellos de botella.	-	*	-	*	-
8	Preparación de datos de ECG.	-	-	*	*	-

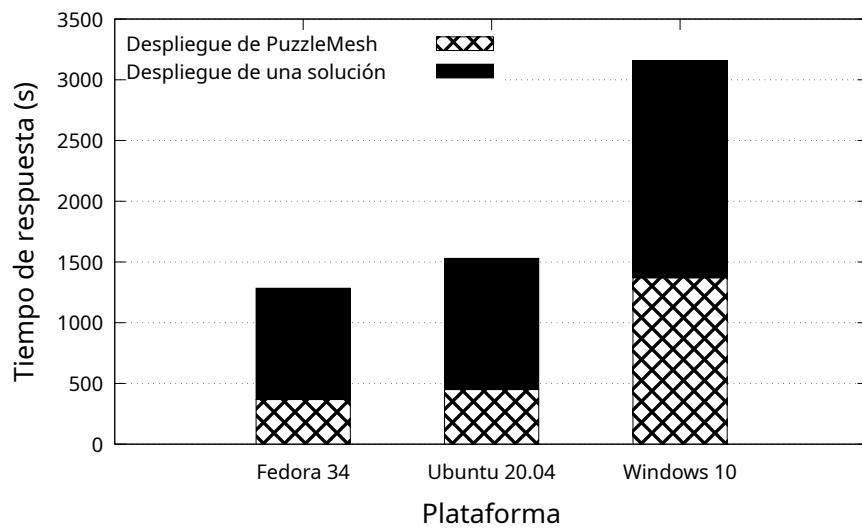


Figura 5.9: Tiempo de respuesta observado durante el despliegue del Rompecabezas 1 en tres diferentes infraestructuras.

Experimento 8. Reutilización de esquemas de preparación para transportar datos de ECG de forma segura.

En la Tabla 5.3 se muestra un resumen de los experimentos previamente descritos, así como los requerimientos no funcionales que estos cumplimentan, haciendo hincapié en los requerimientos considerados en esta tesis para crear soluciones agnósticas de la infraestructura: portabilidad, manejo de la heterogeneidad, reusabilidad, eficiencia y confiabilidad.

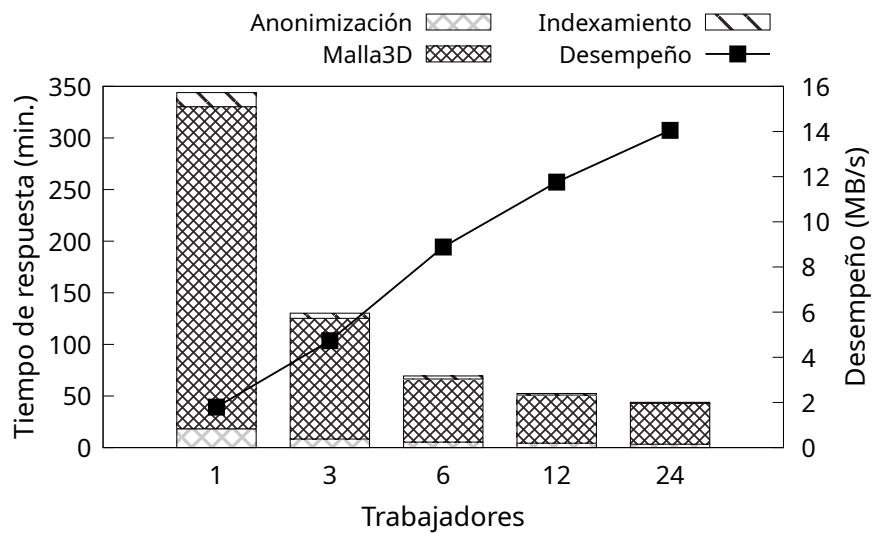


Figura 5.10: Tiempo de respuesta y desempeño observado para el procesamiento de 25 estudios médicos con el Rompecabezas 1

5.5.1.1. Experimento 1: Midiendo los costos de despliegue en diferentes plataformas

En este experimento se evaluó el tiempo de respuesta obtenido durante el despliegue del Rompecabezas 1 en tres diferentes plataformas: Fedora, Ubuntu y Windows. El objetivo de este experimento fue demostrar la portabilidad de las soluciones construidas con PuzzleMesh. En la Figura 5.9 se muestra el tiempo de respuesta obtenido (eje vertical) para desplegar dicha solución en cada plataforma (eje horizontal). El tiempo de respuesta total está compuesto por el tiempo que toma el despliegue de los servicios de construcción de PuzzleMesh, así como la construcción de las piezas consideradas en el rompecabezas y su despliegue. Como se puede observar, Windows es la plataforma en la que se observa el tiempo de respuesta mayor, 52.59 minutos. Mientras que, en Ubuntu y Fedora, el tiempo de respuesta observado fue de 25.49 y 21.39 minutos, respectivamente.

5.5.1.2. *Experimento 2: Evaluación del rendimiento de los patrones paralelos incluidos en PuzzleMesh.*

En este experimento se evaluaron los patrones paralelos incluidos en PuzzleMesh, los cuales permiten mejorar el desempeño de las piezas en un rompecabezas que incluye aplicaciones heterogéneas. En este sentido, se evaluaron dos rompecabezas descritos al inicio de la Sección 5.5, incluyendo diferentes piezas de software para el manejo del ciclo de vida de imágenes médicas.

En la Figura 5.10 se muestra en el eje vertical izquierdo el tiempo de respuesta observado durante el procesamiento de 25 estudios de tomografías de hueso (37 GB de datos) utilizando un número incremental de patrones paralelos (eje horizontal). En el eje vertical derecho de la misma Figura, se observa el desempeño de cada solución medido en megabytes por segundo (MB/s). Como se esperaba, a medida que el número de trabajadores aumenta, el tiempo de respuesta se reduce y el rendimiento aumenta. Por ejemplo, con un trabajador, la fuente de datos es procesada en 352.41 minutos, mientras que con tres trabajadores el tiempo de respuesta es reducido a 133.29 minutos. Como resultado, se observa una ganancia en el tiempo de respuesta de 62.17 % con una aceleración de 2.64x, cuando se comparan ambas soluciones.

Cabe resaltar que el porcentaje de ganancia se reduce a medida que más trabajadores son agregados en el patrón, cuando se compara una configuración con la previa a esta. Por ejemplo, cuando se comparan las configuraciones de tres y seis trabajadores, el porcentaje de ganancia es de 46.62 %. Mientras que cuando se comparan las versiones de seis y doce trabajadores el porcentaje se reduce a 24.76 %. Lo anterior se debe a que los patrones son ejecutados en una infraestructura con 24 núcleos físicos, lo cual limita el número de trabajadores que se pueden desplegar produciendo una ganancia en el tiempo de respuesta.

En la Figura 5.11 se observa, en el eje vertical-izquierdo, el tiempo de respuesta producido por el Rompecabezas 2 cuando se procesan 1104 tomografías de pulmón (580 MB) con un número

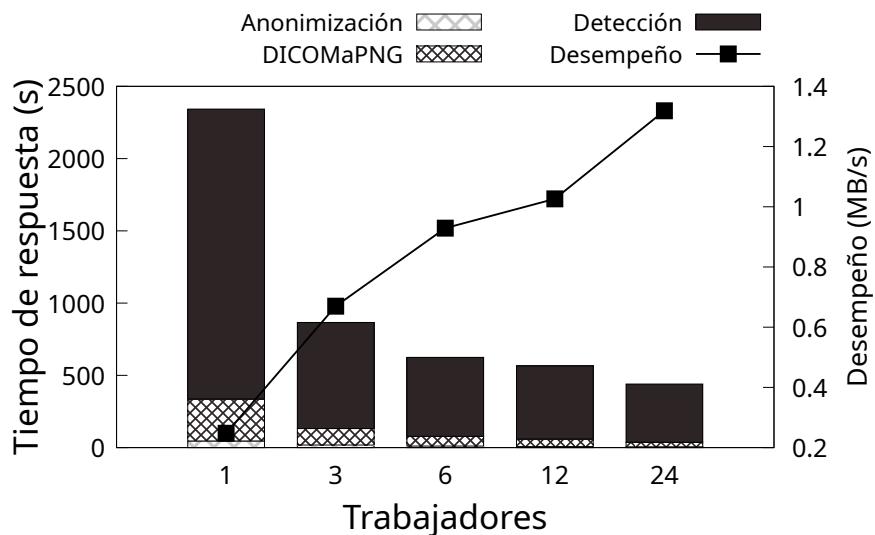


Figura 5.11: Tiempo de respuesta y rendimiento observados durante el procesamiento de 1104 imágenes médicas con un rompecabezas para la identificación de tumores en pulmón.

incremental de trabajadores (eje horizontal). Mientras que en el eje vertical derecho de la misma Figura se observa el rendimiento producido por cada configuración. De nuevo se observa como al aumentar el número de trabajadores en el patrón, el tiempo de respuesta es menor en comparación con las configuraciones con menos trabajadores. Por ejemplo, con un trabajador, el Rompecabezas 2 produjo un tiempo de respuesta de 39.04 minutos, mientras que con seis trabajadores el tiempo de respuesta se redujo a 10.40 minutos. Esto representa una aceleración de 3.75x, con un porcentaje de ganancia de 73.34 %. Cabe resaltar que las tomografías de pulmón fueron clasificadas utilizando solo CPUs, los cuales son más accesibles para las organizaciones médicas que las GPUs.

5.5.1.3. Experimento 3: Análisis estadístico

Para demostrar que la diferencia en rendimiento de las soluciones paralelas evaluadas en la Sección 5.5.1.2 es estadísticamente significativo, se realizó un análisis estadístico utilizando la prueba Kruskal-Wallis [91, 133]. Las hipótesis de esta prueba son:

H₀ Las medianas de los tiempos de ejecución de cada ejecución son iguales.

Tabla 5.4: Resultados de la comparación a pares de cada configuración paralela evaluada.

Configuración 1	Configuración 2	Estadístico T	Valor p	¿Se rechaza la hipótesis nula?
1	3	156.07	≈ 0	Sí
1	6	457.26	≈ 0	Sí
1	12	516.30	≈ 0	Sí
1	24	380.84	≈ 0	Sí
3	6	29.57	≈ 0	Sí
3	12	-41.70	≈ 0	Sí
3	24	-43.49	≈ 0	Sí
6	12	-25.87	≈ 0	Sí
6	24	-28.10	≈ 0	Sí
12	24	7.96	≈ 0	Sí

H1 Las medianas de los tiempos de ejecución de cada ejecución no son iguales.

Como resultado de esta prueba se observó un *valor p* de $2.69e - 30$, dado que este valor es menor que $\alpha = 0.05$ se rechaza la hipótesis nula (H_0), ya que se observa una diferencia estadísticamente significativa entre las medias de las configuraciones evaluadas. Por lo tanto, la ganancia en el tiempo de respuesta observada con cada solución es estadísticamente significativa.

Para complementar este análisis, se realizó un test Post Hoc para conocer las relaciones específicas entre cada configuración evaluada. En la Tabla 5.4 se muestran los resultados obtenidos de realizar la prueba de Bonferroni. Como se puede observar, para cada par de configuraciones se rechaza la hipótesis nula, por lo que existe una diferencia significativa entre el tiempo de respuesta observado en cada solución.

Con estas pruebas se demuestra estadísticamente que el método propuesto logra reducir el tiempo de respuesta, lo cual es crucial en procesos de tomas de decisiones.

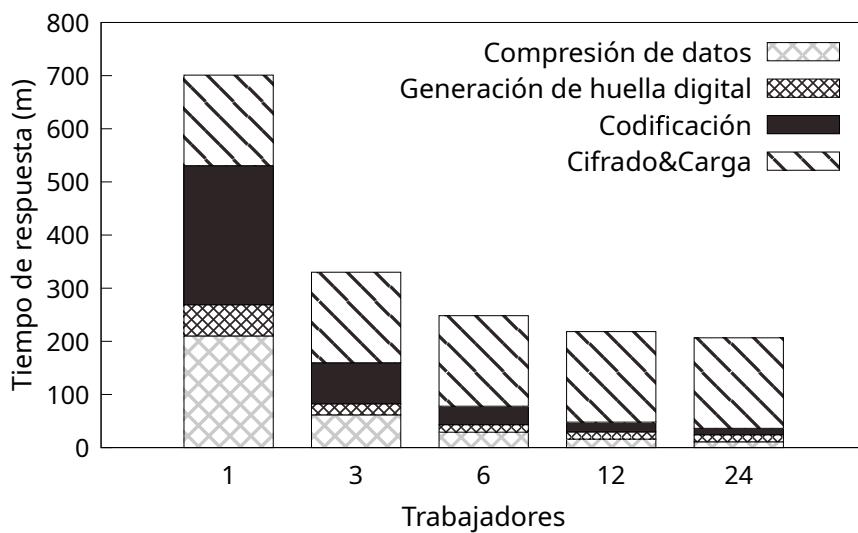


Figura 5.12: Tiempo de respuesta observado al cargar los datos desde el nodo de procesamiento hasta el nodo de almacenamiento.

5.5.1.4. Experimento 4: Analizando los costos de preparación de los datos

Las metas de este experimento son similares a las abordadas en el experimento anterior. En un escenario real, los datos médicos no pueden ser transportados y almacenados sin antes ser preparados para cumplimentar con las normas y regulaciones impuestas para el intercambio y preservación de datos sensibles (p. ej. el cifrado de datos para asegurar la privacidad de la información personal). En este contexto, hemos medido el tiempo de respuesta que toma cargar un conjunto de imágenes médicas (37 GB) a una red de entrega de contenido para su almacenamiento. Antes de cargar los datos, estos fueron comprimidos, indexados en una base de datos (verificando si los datos ya han sido cargados anteriormente), se les agregó información redundante (para soportar errores de integridad), y se utilizaron técnicas de asegurar los datos durante su transporte y almacenamiento.

La Figura 5.12 muestra el tiempo de respuesta (eje vertical) al preparar los datos, variando el número de trabajadores (eje vertical). Como se puede observar, entre más trabajadores sean agregados al patrón, el tiempo de respuesta se reduce. Con un solo trabajador, el tiempo de respuesta para la carga de los 37 GB de datos (70270 archivos) es de 11.68 horas, lo cual se redujo a tan solo 3.44

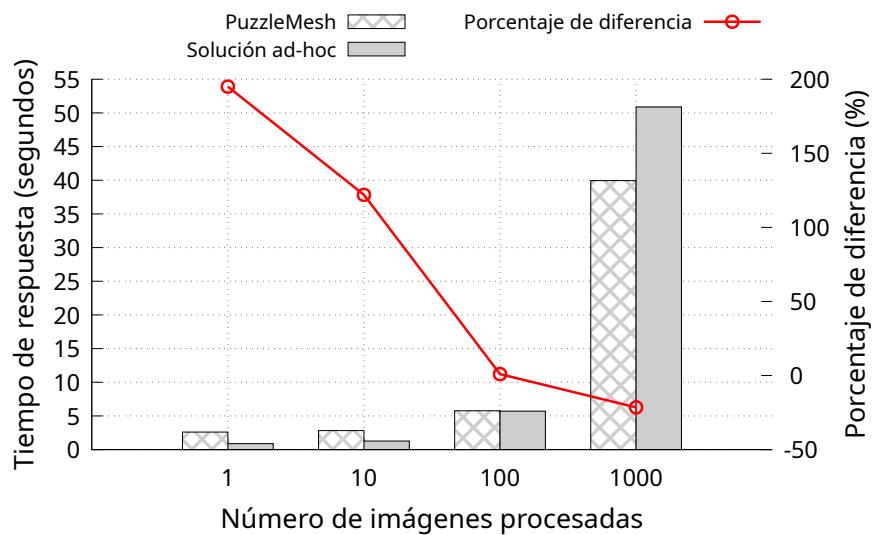


Figura 5.13: Porcentaje de incremento y reducción en el tiempo de respuesta de PuzzleMesh y una solución ad-hoc.

horas con 24 trabajadores. Esto representa una ganancia del 70.52 % en el tiempo de respuesta, con una aceleración de 3.39x.

5.5.1.5. Experimento 5: Comparación con una solución ad-hoc.

En este experimento se realizó una comparación del tiempo de respuesta observado cuando se procesan imágenes médicas con una estructura modular creada con PuzzleMesh y una solución ad-hoc programada en un script de shell, ejecutada directamente desde la consola del sistema operativo CentOS 7. Este script fue diseñado para procesar imágenes médicas con la aplicación de anonimización incluida en el Rompecabezas 1. Para ello, esta aplicación y sus dependencias fueron instaladas y configuradas directamente en el sistema operativo. Por otra parte, en PuzzleMesh este proceso de configuración fue omitido, dado que las piezas autocontenidoas incluyen todas las dependencias y configuraciones requeridas por las aplicaciones para funcionar en diferentes infraestructuras.

En la Figura 5.13 se muestra el tiempo de respuesta (eje vertical-izquierdo) observado en cada

solución para procesar 1, 10, 100 y 1000 imágenes médicas en formato DICOM (eje horizontal). El eje vertical derecho de la Figura 5.13 muestra el porcentaje de incremento o decremento en el tiempo de respuesta de PuzzleMesh en comparación con esta solución ad-hoc. Cabe notar, que en PuzzleMesh no se están procesando datos en paralelo, por lo que ambas soluciones procesan los datos secuencialmente. Como se puede observar, con un número bajo de archivos, el tiempo de respuesta observado en PuzzleMesh es mayor que el producido por la solución ad-hoc. Por ejemplo, el procesamiento de una imagen le tomo a PuzzleMesh 2.60 segundos, mientras que a la solución ad-hoc le tomo 0.88 segundos, lo cual representa una diferencia en el tiempo de respuesta de 195 %. Sin embargo, como se puede observar, a medida que el número de imágenes crece, esta diferencia en el tiempo de respuesta se reduce, e incluso para 1000 imágenes PuzzleMesh produce un tiempo de respuesta es menor. Este conjunto de imágenes es procesado con PuzzleMesh en 39.94 segundos, mientras que la solución ad-hoc procesa las mismas imágenes en 50.87 segundos, lo cual representa una reducción en el tiempo de respuesta de 27.36 %.

Como se puede observar, PuzzleMesh no solo agrega las características de portabilidad a las aplicaciones utilizando piezas autocontenidoas, las cuales permiten el despliegue automático de aplicaciones, sino que además permite el manejo eficiente de volúmenes grandes de productos digitales, reduciendo el tiempo de respuesta observado por los usuarios finales, en comparación con una solución ad-hoc.

5.5.1.6. Experimento 6: Comparación del rendimiento de PuzzleMesh con una herramienta de la literatura

En este experimento realizamos una comparación directa del rendimiento de PuzzleMesh con el rendimiento de Makeflow, el cual es un motor de flujos de trabajo encontrado en la literatura. Para este experimento, el Rompecabezas 1 fue implementado utilizando PuzzleMesh y Makeflow. La Figura 5.14 muestra el tiempo de respuesta en minutos (eje vertical) producido por cada una de

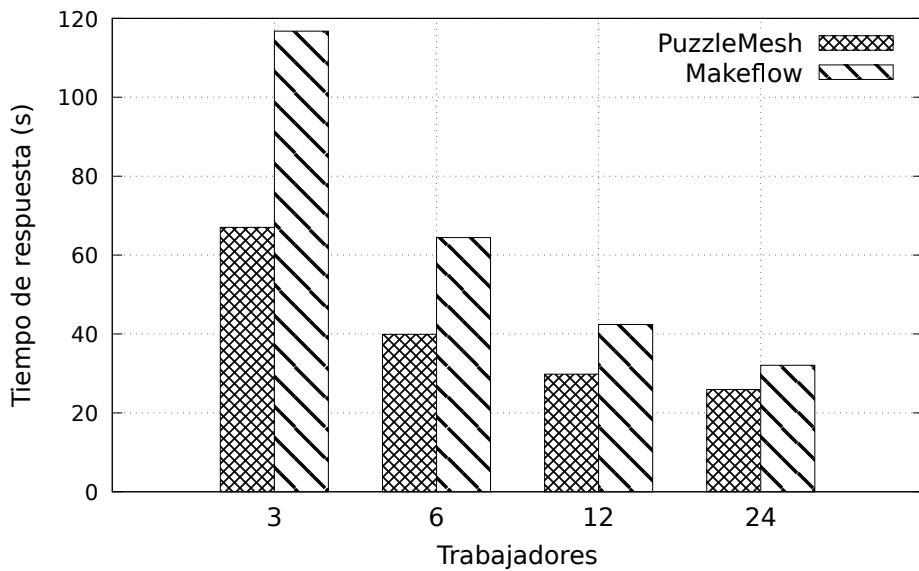


Figura 5.14: Comparación directa de PuzzleMesh con Makeflow.

las soluciones, utilizando diferente número de trabajadores paralelos (eje horizontal) o subprocessos paralelos. Como se puede observar en la Figura 5.14, PuzzleMesh produce un menor tiempo de respuesta para cada configuración. PuzzleMesh reduce el tiempo de respuesta en comparación con Makeflow en un 42.57 %, 38.06 %, 29.68 %, y 19.17 % para 3, 6, 12, y 24 trabajadores, respectivamente. Es importante notar que entre más trabajadores son agregados, la diferencia en el tiempo de respuesta entre cada una de las soluciones se reduce. No obstante, a partir de los resultados presentados en la Figura 5.14 es posible observar que PuzzleMesh produce un tiempo de respuesta menor que Makeflow utilizando un menor número de trabajadores (recursos). Por ejemplo, con 6 trabajadores, PuzzleMesh procesa los datos en 39.91 minutos, mientras que Makeflow procesa la misma cantidad de datos en 42.40 minutos utilizando 12 trabajadores.

5.5.1.7. Experimento 7: Manejo de cuellos de botella en tiempo de ejecución

En este experimento se evaluó cómo PuzzleMesh es capaz de manejar y mitigar cuellos de botella en una estructura de procesamiento durante tiempo de ejecución. En este experimento se evaluaron

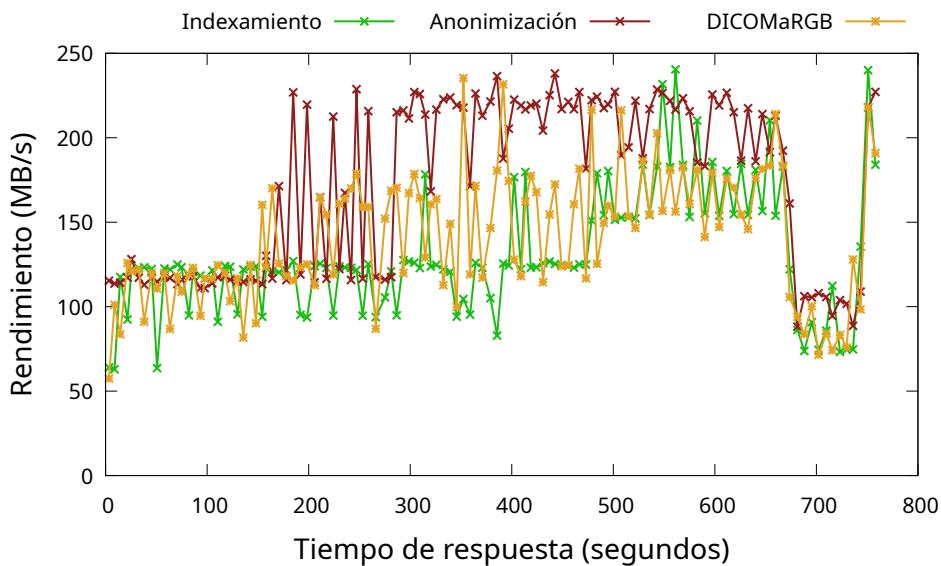


Figura 5.15: Rendimiento observando durante el procesamiento de imágenes médicas con 3 piezas.

las siguientes soluciones:

Tres piezas. Rompecabezas que incluye tres piezas para el manejo de datos médicos, considerando las piezas de indexamiento de datos, anonimización y conversión de DICOM a RGB.

Cuatro piezas. Este rompecabezas incluye las tres piezas anteriores y se incluye una cuarta para la identificación de tumores en tomografías de pulmón.

- La pieza de Anonimización es aquella que da un rendimiento mayor al resto, por lo que las piezas de Indexamiento y Conversión de DICOM a RGB son identificadas como cuellos de botella en diferentes puntos de la ejecución. Por ejemplo, la primera medición (a los 3 segundos de ejecución) tomada para la pieza de Indexamiento es de 63.96 MB/s y para la de Conversión es de 57.45 MB/s, mientras que para la de Anonimización es de 115.35 MB/s.
- Al parallelizar los cuellos de botella durante tiempo de ejecución, el rendimiento de estas aumenta. Por ejemplo, a los 14 segundos de ejecución, el rendimiento de las piezas de Indexamiento y Conversión aumentó a 117.59 MB/s y 83.63 MB/s, respectivamente. Esto representa un incremento en el rendimiento de 30.77% y 54.35%, para ambas piezas

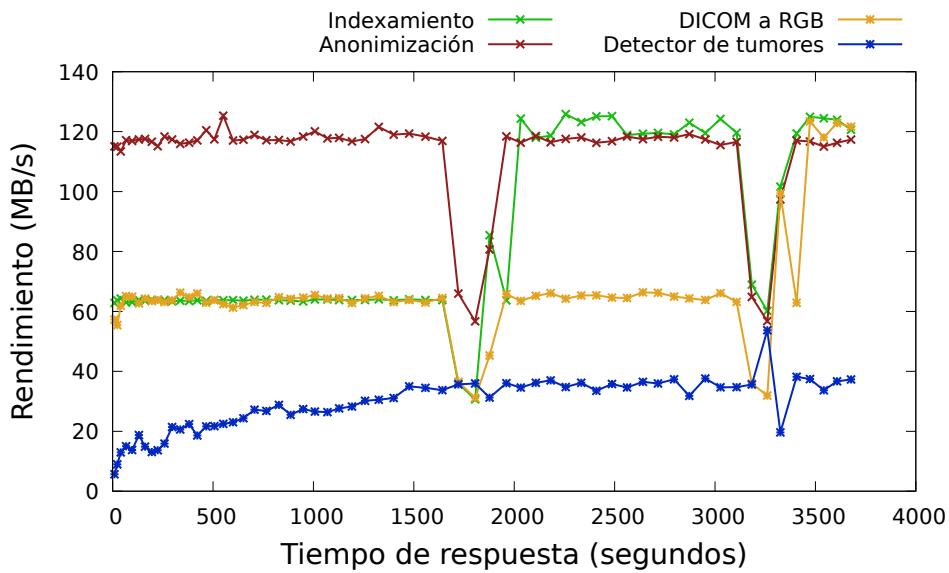


Figura 5.16: Rendimiento observando durante el procesamiento de imágenes médicas con 4 piezas.

respectivamente.

- El rendimiento de las piezas se homogeneiza para las tres piezas, lo cual permite generar un flujo de datos continuo.

En la Figura 5.16 se muestran los resultados observados para el manejo de cuellos de botella en un rompecabezas con cuatro piezas. En este caso, el cuello de botella observado durante tiempo de ejecución es la pieza de Detección de tumores. Como se puede observar, el rendimiento de esta pieza aumenta durante tiempo de ejecución, pasando de 5.64 MB/s a 53.72 MB/s en su punto máximo. Esto representa un incremento de 89.50 % en el desempeño. De los resultados observados en la Figura 5.16, podemos concluir que, al existir un cuello de botella con un rendimiento considerablemente inferior al resto, PuzzleMesh intenta mitigarlo hasta llegar a un punto en el que el rendimiento no puede crecer más, debido a las limitaciones en la infraestructura.

Como se puede observar, este modelo de mitigación de cuellos de botella es capaz de generar soluciones paralelas durante tiempo de ejecución sin intervención del diseñador y el usuario final. Esto es útil cuando los diseñadores no conocen la carga de trabajo entrante ni el comportamiento

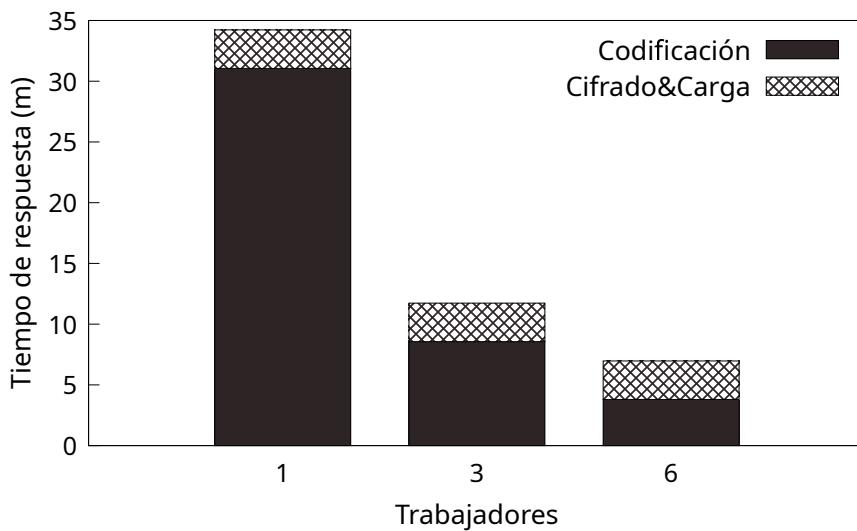


Figura 5.17: Tiempo observado durante la codificación cifrado y carga de 1000 trazas de ECG.

de sus aplicaciones, por lo que requeriría de una etapa de ajuste y experimentación para identificar cuellos de botella y la configuración de paralelismo que los mitigue.

5.5.1.8. *Experimento 8: Reutilizando los esquemas de preparación de datos para transportar datos de ECG.*

Este experimento se condujo para demostrar la flexibilidad de las piezas de PuzzleMesh para ser utilizadas para construir otros sistemas de aplicaciones. En este caso, se reutilizaron los esquemas de preparación para preparar 1000 trazas de ECG (391 MB), añadiendo las características de confiabilidad y seguridad a los datos.

En el eje vertical de la Figura 5.17 se observa el tiempo de respuesta, en minutos, obtenido para la preparación de 1000 trazas de ECG utilizando un número incremental de trabajadores (eje horizontal). De nuevo, se observa que el tiempo de respuesta se reduce a medida que nuevos trabajadores son agregados en los patrones. Por ejemplo, con 1 trabajador el tiempo de respuesta observado es de 34.22 minutos, mientras que con seis trabajadores el tiempo de respuesta se reduce a 6.96 minutos, lo cual representa una ganancia en el tiempo de respuesta de 79.63 %.

5.6 Análisis de la experiencia de usuario

En esta sección se compara la experiencia de usuario al crear diferentes soluciones con tres motores de flujo de trabajo (ParSl, Pegasus y Makeflow) y PuzzleMesh. La comparación considera todas las acciones que un desarrollador tiene que realizar crear una estructura de procesamiento.

En la Figura 5.18 se muestran estas acciones organizadas como tuberías. En esta Figura, cada acción realizada en tiempo de configuración, desarrollo y ejecución son mostrados con diferentes símbolos. En tiempo de configuración se consideran aquellas acciones que el desarrollador debe de realizar para instalar correctamente la herramienta o motor de flujos de trabajo a utilizar. Mientras que, en tiempo de desarrollo, se considera la creación de la estructura de procesamiento mediante un modelo de programación o un método declarativo. Finalmente, en tiempo de ejecución se consideran las acciones que realiza la herramienta para manejar la estructura de procesamiento.

Además, en la Figura 5.18 las cajas largas representan acciones realizadas manualmente por el desarrollador, mientras que las cajas cortas representan acciones que son realizadas automáticamente por la herramienta.

Como se puede observar, PuzzleMesh es la herramienta que no solo incluye el mayor número de acciones automáticas, sino que además reduce el número de acciones requeridas para configurar y diseñar sistemas de aplicaciones, así como que no requiere la instalación de herramientas de terceros para realizar la ejecución de estas estructuras.

PuzzleMesh considera piezas de software para el manejo transparente e implícito de paralelismo, balanceo de carga, acoplamiento y ejecución de aplicaciones, así como el manejo, entrega y recuperación de datos durante tiempo de ejecución. Las estructuras autocontenidoas y autoadaptables de PuzzleMesh, convierten una aplicación en una pieza de software independiente que mejora la usabilidad de las soluciones, dado que estas están listas para usarse y desplegarse.

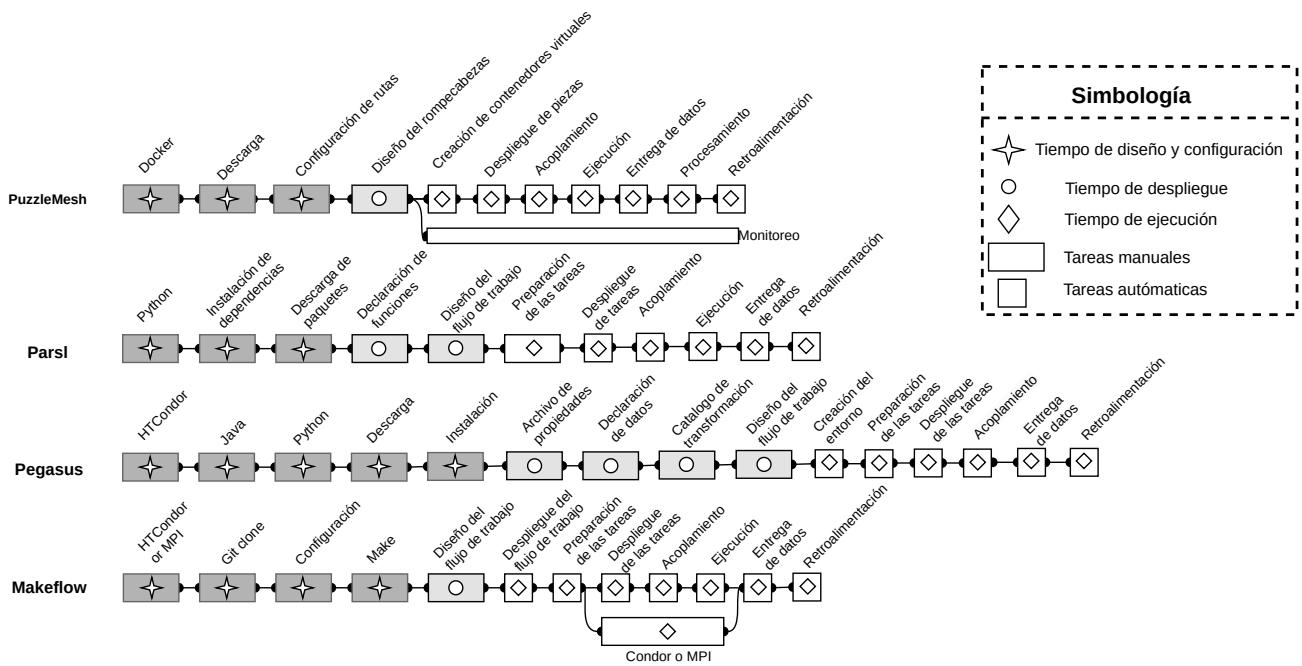


Figura 5.18: Análisis comparativo de las acciones requeridas para configurar, desarrollar y ejecutar una solución utilizando diferentes herramientas para construir sistemas de aplicaciones.

Estos bloques independientes mejoran la flexibilidad para crear diferentes patrones de paralelismo en cada pieza de una estructura de procesamiento. Esto significa que cada pieza puede ser configurada para manejar un número diferente de trabajadores paralelos, dependiendo los recursos disponibles en cada pieza, lo cual mejora el manejo de los trabajadores en paralelo para mitigar cuellos de botella.

6

Conclusiones, limitaciones y trabajo futuro

En el presente Capítulo se exponen las principales conclusiones obtenidas de este trabajo de investigación, así como las limitaciones identificadas durante el diseño, desarrollo y evaluación del método propuesto. Además, se presenta el trabajo futuro que permitirá contemplar este trabajo de investigación.

6.1 Conclusiones

En esta tesis se describió el diseño, implementación y evaluación de un método de procesamiento para construir y manejar estructuras de procesamiento agnósticas de la infraestructura durante su ciclo de vida, en tiempos de diseño, despliegue y ejecución.

La evaluación, tanto cualitativa como cuantitativa, mostró que el método propuesto en este proyecto de investigación efectivamente permite a los usuarios crear, a alto nivel, estructuras de procesamiento

de diferentes tipos (no solo tuberías) usando múltiples aplicaciones desarrolladas utilizando diferentes tipos de lenguajes de programación y que, en forma transparente y automática, estas estructuras se despliegan sobre múltiples infraestructuras.

El modelo de construcción llamado PuzzleMesh, basado en bloques autosimilares y autocontenido, permite a los diseñadores encapsular sus aplicaciones en bloques llamados *piezas*. Las *piezas* son artefactos de software que contienen interfaces de entrada y salida, estructuras de acoplamiento con otras piezas, patrones de preparación de datos, balanceadores de carga y esquemas de paralelismo basados en patrones paralelos. Lo anterior permite a los diseñadores crear piezas de software que pueden ser inmediatamente desplegadas en múltiples infraestructuras (por ejemplo, en la nube o un clúster HPC) y acopladas con otras piezas, aun cuando estas implementen aplicaciones desarrolladas en otros lenguajes de programación (manejo de la heterogeneidad). Debido a que estas piezas son autocontenido, pueden ser desplegadas en diferentes infraestructuras (portabilidad) sin tener que hacer cambios en el código fuente de la aplicación, reduciendo o eliminando las tareas requeridas para desplegar la aplicación en la nueva infraestructura (p. ej. configuración de dependencias y variables de entorno). La característica de autosimilitud de las piezas permite que estas, algunos de sus componentes o incluso una estructura de procesamiento sea reutilizada para crear una nueva solución.

Este modelo ha permitido responder a la primera pregunta de investigación sobre *¿Cómo gestionar la heterogeneidad en el diseño de las estructuras de procesamiento utilizadas para crear sistemas de aplicaciones para la gestión de CVPD?* Esto también nos ha permitido conseguir el objetivo de *crear un modelo de construcción basado en bloques autosimilares y autocontenido para gestionar la heterogeneidad de las estructuras de procesamiento utilizadas para el manejo del CVPD*.

La evaluación también ha mostrado que un modelo de comunicación y programación de infraestructura cómo código permite realizar eficiente, eficaz, transparente y automáticamente, el acoplamiento de diferentes piezas desplegadas en múltiples infraestructuras. Esto nos ha permitido

conseguir el objetivo de *crear y definir un modelo de comunicación y programación de infraestructuras como código para materializar y desplegar, en forma transparente y automática, estructuras de procesamiento sobre múltiples infraestructuras.*

La implementación de estos dos modelos en un prototipo nos permitió dar respuesta a la pregunta sobre *¿Cómo construir estructuras de procesamiento que permitan reducir, en tiempo de despliegue, las potenciales dependencias de las estructuras de procesamiento con el lenguaje de programación, la plataforma o la infraestructura?* Ya que el modelo de comunicación realiza eficazmente el acoplamiento de las piezas de software, mientras que el modelo procesamiento se encarga de crear flujos de datos a través de las piezas siguiendo un modelo ETL y utilizando una red de distribución de contenidos para transportar los datos entre piezas. Esto ha permitido mostrar, en múltiples estudios de caso basados en sistemas de ciencia desplegados sobre múltiples tipos de infraestructuras, que los sistemas de aplicaciones consiguen desplegar, ejecutar y crear exitosamente, en todos los casos, los flujos de datos diseñados en alto nivel. Estos modelos, aplicados en la fase de despliegue del método, muestran que los sistemas de aplicaciones evitan dependencias (software, plataforma e infraestructura), ya que se autoacoplan y procesan los CVPDs.

Finalmente, la evaluación de los múltiples estudios de caso demostró que los sistemas de aplicaciones se han mantenido procesando datos incluso en escenarios de sobrecargas de trabajo, los cuales son producidos por cargas de trabajo y rendimiento de recursos computacionales desconocidos mediante la implementación de un modelo de gestión de tareas/datos y paralelismo implícito integrado a un esquema de mitigación de cuellos de botella embebido en las estructuras de procesamiento. Este modelo, basado en un esquema de monitoreo y manejo de colas, y un mecanismo de autoescalamiento nos ha permitido dar respuesta a la pregunta sobre *¿Cómo proveer a las estructuras de procesamiento con mecanismos que permitan reducir los cuellos de botella que se puedan crear en tiempo de ejecución para mitigar los efectos de escenarios de saturación?* Esto, debido a que la implementación e inclusión de estos modelos en los esqueletos en los que se montan los sistemas de aplicaciones para

CVPD permite mantener el procesamiento de datos incluso en dos tipos de escenarios que podrían producir una interrupción del flujo de producción de productos digitales: *i)* fallas en la disponibilidad de los datos, y *ii)* el manejo de escenarios de saturación, los cuales son los más frecuentemente encontrados en escenarios de ciencia de datos.

En el primer escenario, los sistemas logran identificar fallas en el acceso a los datos, ya que los esqueletos incluyen patrones de preparación que implementan algoritmos de dispersión de información y seguridad informática para tolerar fallas en el acceso a los datos enviados a los nodos de almacenamiento de datos. En escenarios donde se detecte que los datos no se encuentran íntegros o simplemente no se pueden acceder, los datos se reconstruyen automáticamente.

En el segundo escenario, el método de monitoreo no solo identifica fallas en el acceso a los datos, sino que también recolecta métricas de desempeño de las piezas en una estructura de procesamiento, lo cual permite identificar cuellos de botella que generan retrasos afectando los tiempos de respuesta. En este sentido, para mitigar estos cuellos de botella se implementó un esquema de autoescalamiento que tiene como objetivo distribuir la carga entrante entre los diferentes trabajadores disponibles, manejando así las colas de entrada de datos.

La implementación de estos modelos, esquemas y mecanismos en el esqueleto de los sistemas de aplicaciones nos ha permitido conseguir el objetivo de *crear un modelo de gestión de tareas/datos y paralelismo implícito para mitigar, en tiempo de ejecución, los efectos de cuellos de botella y evitar la interrupción de procesamiento en escenarios de saturación en las estructuras de procesamiento*.

Además, la implementación del método propuesto en una plataforma orientada al diseño ha sido utilizada para el desarrollo de sistemas de ciencia de datos puestos en producción en un hospital, así como en otras instituciones de salud pública. Esto nos ha permitido observar que, a partir de un diseño en alto nivel (basado en el acoplamiento de piezas que permite la construcción de múltiples patrones, y las cuales son elegidas por los usuarios finales) la plataforma convierte este diseño en una estructura de procesamiento, la cual incorpora el sistema de aplicaciones en un esqueleto agnóstico

que es automáticamente desplegado sobre las infraestructuras declaradas por los usuarios. Todo lo anterior es realizado en forma automática y en un solo proceso de construcción transparente. Esto nos ha permitido obtener el objetivo general de este proyecto de tesis, el cual es *crear un método de composición de servicios para diseñar, desplegar y gestionar la operación continua de estructuras de procesamiento agnósticas de la infraestructura*. Lo anterior nos ha permitido comprobar nuestra hipótesis donde se estableció que la propiedad de agnosticismo agregado a las estructuras de procesamiento en la forma de un esqueleto embebido a los sistemas de aplicaciones resolvería las dependencias funcionales que se presentan en tiempos de diseño, despliegue y ejecución de los sistemas de aplicaciones (e.g. sistemas de ciencia de datos o sistemas de big data).

Hallazgos sobre el impacto de la implementación del método sobre el rendimiento de los sistemas de aplicaciones

La evaluación experimental del método propuesto consideró la conducción de estudios de caso basados en el manejo de imágenes satelitales, contenidos médicos y registros meteorológicos. La evaluación experimental reveló la eficiencia del método propuesto en comparación con soluciones del estado del arte, mejorando los tiempos de respuesta, lo cual es crucial durante el manejo de productos digitales y en procesos de toma de decisiones. Para demostrar las características de portabilidad, reusabilidad y manejo de la heterogeneidad se condujeron diferentes experimentos utilizando diferentes infraestructuras, aplicaciones y estructuras de procesamiento.

En un estudio de caso para el procesamiento de imágenes satelitales, se observó un mejor desempeño de PuzzleMesh en comparación con otras herramientas del estado del arte. En este sentido, el método propuesto produjo una ganancia en el tiempo de respuesta del 16.81 %, 21.39 %, 25.26 % y 31.26 % en comparación con DagOnStar, Parsl, Makeflow y Jenkins, respectivamente.

De forma similar, en un estudio de caso para el procesamiento de datos médicos, se observó que

el método propuesto en comparación con Makeflow produjo una ganancia del 42.57 %, 38.06 %, 29.68 %, y 19.17 % utilizando configuraciones de 3, 6, 12, y 24 trabajadores, respectivamente.

En un análisis de la experiencia del usuario, al utilizar PuzzleMesh y tres motores de flujo de trabajo (ParSl, Pegasus y Makeflow), se observó que el método propuesto reduce el número de pasos requeridos para la configuración, diseño, despliegue y ejecución de estructuras de procesamiento al reducir/eliminar la dependencia de estas con herramientas con terceros.

Además, en diferentes análisis cualitativos se mostró que PuzzleMesh permite no solo el diseño de estructuras de procesamiento, sino que también es posible manejar diferentes requerimientos no funcionales que son cruciales en el manejo de datos y aplicaciones, tales como eficiencia, confiabilidad, escalabilidad, portabilidad, manejo de la heterogeneidad y seguridad. Lo anterior no suele ser contemplado en diferentes motores de flujos de trabajo o gestores de tuberías, los cuales se encuentran centrados en el procesamiento y entrega de datos.

Finalmente, a partir de los resultados de experimentación se validó la hipótesis de esta investigación, debido a que se obtuvo un método de composición de servicios que permite a las organizaciones manejar servicios agnósticos de la infraestructura con los cuales pueden manejar el ciclo de vida de sus productos digitales. Además, que la eficiencia del método propuesto es superior a herramientas de la literatura, y la evaluación demostró la flexibilidad del método para manejar diferentes ciclos de vida.

6.2 Limitaciones

Actualmente, el modelo de mitigación de cuellos de botella utiliza esquemas de autoescalamiento en los cuales se agregan o eliminan trabajadores de una pieza/etapa para mitigar los posibles cuellos de botella. En específico, estos esquemas utilizan escalamiento vertical para agregar nuevos trabajadores en la etapa identificada como cuello de botella. En este sentido, el grado de escalamiento de una pieza

depende de la infraestructura disponible, especialmente del número de núcleos con los que cuente el equipo. Durante la experimentación, se observó que el porcentaje de ganancia en los tiempos de respuesta se reduce a medida que el número de trabajadores se acerca al número de núcleos físicos con los que cuenta el equipo. Por lo tanto, se pueden explorar otras opciones como manejo en memoria de los buffers de entrada y salida de las piezas/etapas, así como la implementación de técnicas de escalamiento horizontal.

Actualmente, el método considera el uso de contenedores virtuales para agilizar el despliegue de aplicaciones y servicios. En la literatura se ha documentado que esta tecnología puede tener fallos de seguridad cuando no se configuran correctamente, permitiendo a terceros tener acceso de superusuario a la infraestructura donde están desplegados [9]. Por lo tanto, en algunos escenarios no sería deseable el uso de contenedores virtuales o se tendrían que implementar esquemas de auditoría de contenedores virtuales para garantizar que su despliegue no producirá brechas de seguridad.

6.3 Trabajo futuro

Como trabajo futuro de esta investigación se planea analizar requerimientos no funcionales como seguridad, trazabilidad y disponibilidad para agregarlos a los servicios agnósticos de la infraestructura. Además, se explorarán diferentes mecanismos para la mitigación de cuellos de botella, como lo es el manejo de la memoria disponible para cada etapa y la implementación de patrones de paralelismo autoadaptables. Finalmente, se examinarán diferentes opciones para crear ambientes de ejecución segura dentro de los contenedores virtuales.

A

Publicaciones

Como parte de este trabajo se publicaron los siguientes artículos de revista, en donde se describen los principales resultados obtenidos en esta tesis.

- En el siguiente artículo se presenta una primera evaluación del modelo de construcción basado en bloques autosimilares y autocontenidos, el cual forma parte del Objetivo 1 de esta tesis:
 - Sánchez-Gallegos, D. D., Di Luccio, D., Gonzalez-Compean, J. L., & Montella, R. (2019, November). A microservice-based building block approach for scientific workflow engines: Processing large data volumes with dagonstar. In 2019 15th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS) (pp. 368-375). IEEE.
- En el siguiente artículo se describe el diseño e implementación de un modelo para permitir el despliegue de estructuras de procesamiento en múltiples infraestructuras, lo cual forma parte del Objetivo 2 de esta tesis:

- Sánchez-Gallegos, D. D., Di Luccio, D., Kosta, S., Gonzalez-Compean, J. L., & Montella, R. (2021). An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience. *Future Generation Computer Systems*, 122, 187-203.
- En el siguiente artículo se describe un modelo para manejar múltiples estructuras de procesamiento agnósticas de la infraestructura y mitigar cuellos de botella utilizando patrones de paralelismo, lo cual forma parte del Objetivo 3 de esta tesis:
 - Sanchez-Gallegos, D. D., Gonzalez-Compean, J. L., Carretero, J., Marin, H., Tchernykh, A., & Montella, R. (2022). PuzzleMesh: A puzzle model to build mesh of agnostic services for edge-fog-cloud. *IEEE Transactions on Services Computing*.

A continuación se anexan los artículos anteriormente mencionados.

A Microservice-Based Building Block Approach for Scientific Workflow Engines: Processing Large Data Volumes with DagOnStar

Dante D. Sánchez-Gallegos
Unidad Tamaulipas, Cinvestav
 Victoria, Mexico
 dsanchez@tamps.cinvestav.mx

Diana Di Luccio
Dpt. of Science and Technologies
University of Naples “Parthenope”
 Naples, Italy
 diana.diluccio@uniparthenope.it

J. L. Gonzalez-Compean
Unidad Tamaulipas, Cinvestav
 Victoria, Mexico
 jgonzalez@tamps.cinvestav.mx

Raffaele Montella
Dpt. of Science and Technologies
University of Naples “Parthenope”
 Naples, Italy
 raffaele.montella@uniparthenope.it

Abstract—The impact of machine learning algorithms on everyday life is overwhelming until the novel concept of datacracy as a new social paradigm. In the field of computational environmental science and, in particular, of applications of large data science proof of concept on the natural resources management this kind of approaches could make the difference between species surviving to potential extinction and compromised ecological niches. In this scenario, the use of high throughput workflow engines, enabling the management of complex data flows in production is rock solid, as demonstrated by the rise of recent tools as Parsl and DagOnStar. Nevertheless, the availability of dedicated computational resources, although mitigated by the use of cloud computing technologies, could be a remarkable limitation. In this paper, we present a novel and improved version of DagOnStar, enabling the execution of lightweight but recurring computational tasks on the microservice architecture. We present our preliminary results motivating our choices supported by some evaluations and a real-world use case.

Index Terms—Microservices, Workflows, Virtual Containers, Parallel Processing, Cloud Computing

I. INTRODUCTION

The acquisition, processing, management, and discovery of data are key tasks when observing the earth and its natural phenomena. These tasks also produce information for organizations to support the decision-making process and to conduct scientific studies. The applications that perform these tasks are commonly modeled as stages, which are chained to create processing structures called workflows [1], [2]. These structures are created by using directed acyclic graphs (DAGs), where nodes represent processing stages and the edges represent the I/O interfaces used to chain the stages considered in a workflow.

Multiple tools and frameworks are currently available for scientists to design workflows [1]–[3]. These tools are in charge of executing and deploying each stage of a workflow, in an automatic manner, on a high-performance IT infrastructure

(e.g., clusters of servers, the cloud or HPC [3], [4]). These tools are also in charge of extracting data required by the stages to transform data into information as well as delivering that information either to another stage according to a well-defined sequence (DAG) in a synchronized manner.

The traditional scientific workflows include multiple heterogeneous tasks created by using diverse types of programming models, which even could be executed in several platforms and/or infrastructures. Moreover, the tasks may be deployed on either distributed or centralized IT infrastructures such as high-performance clusters (HPC), the cloud and single servers [2], [5]. Nevertheless, traditional workflows require a homogeneous environment to ensure a given functionality. For instance, all the applications should be written in a given programming language (e.g., Python [1], [2] and Java [6]), and for a given platform (e.g., a given operating system). These restrictions are commonly imposed to enforce the correct functionality of the parallel model (commonly based on multi-threads at stage level) added to the workflow frameworks for improving the performance of workflow solutions created by scientists. This heterogeneity also produces performance issues affecting the efficiency of the workflow solutions.

In this context, there is a need for solutions that not only enable scientists to organize their applications in the form of workflows and execute them in an automatic manner, but also for solutions that can deal with the heterogeneity of the development and performance of applications included in workflows that arises in a real-world scenario.

In this paper, we present the design, development, and evaluation of a building block approach for workflow engines by using virtual containers (VCs) and microservices. In this approach, the building blocks encapsulate the stages of the workflows into VCs to provide the stages with portability. These building blocks are interoperable software pieces, which

can be organized in the form of parallel patterns to improve the performance of the stages encapsulated into the building blocks and to solve the performance heterogeneity of the stages of a workflow. These building blocks are managed in the form of microservices to ensure the correct functionality of the parallel model as well as to solve the development heterogeneity issues that arise when managing stages developed by using different programming languages.

To show the feasibility of this approach, we added the building block management in a microservice architecture to a workflow engine called DagOnStar [5]. To show the efficiency of this approach, we developed a prototype with the new version of DagOnStar, which was evaluated in two case studies. The first case is based on workflows that extract real repositories of environmental data collected from IoT Sensors. These data are transformed through stages such as data pre-processing to detect outliers, data processing to find out hidden patterns in the data, and data storing for preserving the processed data.

The second case was conducted by pre-processing and processing satellite imagery captured by a sensor called LandSat8 to correct radiometric and atmospheric issues in the satellite images of this repository.

In these case studies, the workflows were created by using the building blocks of the DagOnStar prototype. The costs of the workflow deployment and the performance of the resultant workflows were analyzed. Moreover, a comparison of this prototype with engines available in state of the art was also conducted.

The evaluation revealed the feasibility of using a building block approach to deploy workflows on a microservice architecture in an automatic manner, which solves the development of heterogeneity issues. It also revealed the flexibility of this approach to improve the performance of the workflows created with this approach by using master/slave patterns based on building blocks, which solving the performance heterogeneity issues.

The outline of this paper is as follows. In Section II related work to our solution is discussed. Section III is about design strategies of the building blocks. Section IV describes the implementation of a prototype based on the proposed approach. Section V the evaluation run to test DagOnStar. Section VI presents the first case study focused on the processing of IoT sensors. Section VII presents the second case study focused on the processing of satellite imagery. Finally, Section VIII gives conclusion remarks and draws the path for future research development.

II. RELATED WORK

The workflow systems enable organizations to deploy pipelines of applications on different types of IT infrastructures. Galaxy [1], Parsl [2], Pegasus [7], Makeflow [8] and Swift [9] are only some examples of this type of system.

In practice, it is common that IT staff performs troubleshooting procedures when the organizations deploying workflows on a given infrastructure (e.g., clusters of computers, virtual

machines in the cloud, and the grid). This type of procedure includes tasks such as installing applications and dependencies in virtual/physical machines, setting the environment of each application and arrangements in the configurations of each infrastructure to deploy virtual machines. Troubleshooting results in downtime that could disturb the continuity of studies and the usage of virtual machines could increase the time spent in deploying workflows.

The Virtual Containers (*VCs*) [10] have become an alternative to virtual machines as this virtualization technique reduces the need for troubleshooting procedures and the deployment time. Therefore, workflows engines, such as Skyport [11], Taverna [12], and Makeflow [8], are currently using virtual containers for end-users to deploy workflow solutions on different types of platforms. Scripting languages (e.g., YAML [13] or Swift [9]) have also been proposed to create workflows by using codification.

Nevertheless, the heterogeneity of the development and performance of the applications is an open issue for traditional workflow engines. Moreover, the reutilization of smaller parts of a workflow (a task or a subset of tasks) is not trivial because of the dependency between tasks.

The microservice architectures [14] have become a popular technology that allows developers to decouple modules from a large service to create tiny independent and isolated services [15]. The microservices already allow users to create different flows by re-using microservices previously created [15] and reduce the effects of the deployment application issues.

Instead of using multi-thread parallelism as used in traditional workflow engines [5], [16], the approach proposed in this paper is based on parallel patterns created by using containerized building blocks, which are focused on addressing the performance heterogeneity issue. In addition, the incorporation of microservices has been proposed for facing up heterogeneity application deployment and enabling the re-use of components.

III. DESIGN PRINCIPLES OF THE BUILDING BLOCK-BASED APPROACH

In this section, we present the design principles of a Microservice-based building block approach for scientific workflow engines.

A. *DagOnStar* Overview

We designed DagOnStar [5] to reduce the runtime footprint within the actual application. DagOnStar leverages the application life-cycle, supported by a Python library providing the main system components, while a not mandatory service component is used for workflow monitoring and management.

Figure 1 shows a conceptual representation of DagOnStar architecture. As it can be seen, this architecture considers four main components: i) The *Python library ecosystem* that supports the workflow building. ii) The *workflow engine* maps applications with tasks, creates stages, assigns tasks to stages and builds workflows, including the created stages by using a directed acyclic graph (DAG). Service such as DagOnStar

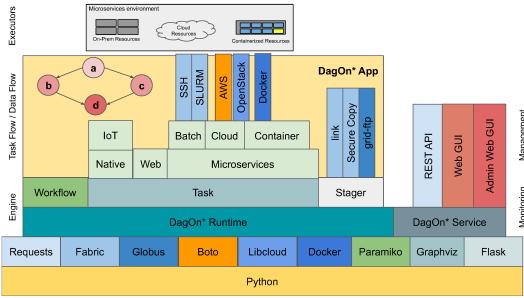


Fig. 1. The DagOnStar architectural schema. Tasks represented as container scripts and executed on a *containerized* infrastructure.

Runtime [17] and DagOnStar Service are in charge of the management and monitoring of building procedures. iii) *Task/data flow management* considers a suite of APIs to prepare the environment (basically defining configurations) according to the DAG of a workflow. iv) The *Executors* are in charge of the workflow deployment on a given infrastructure (e.g., Cloud services, container-based cloud or On-Premise Resources). Executors also establish controls and coordination over the execution of the tasks in the infrastructure (see Figure 1). A DagOnStar application is developed as a Python script where parallel tasks are defined by using the **Task** class. The DagOnStar Runtime performs the interaction with the executors both on local or remote resources instanced by either virtual machines or VCs deployed on public/private/hybrid clouds.

The DagOnStar design considers the `workflow://` schema as the root of the current workflow virtual file system. Under these conditions, `workflow:///workflow_unique_name///task_unique_name/` is the root of the scratch directory created by the DagOnStar Runtime. DagOnStar uses this notation to evaluate the task dependencies using a back-reference approach.

An important issue in this type of solution is data management. In DagOnStar, dataflows are constructed by using the `workflow://` schema, which indicated the data dependencies between tasks.

If two or more workflows interact with each other (regardless if they belong to the same application or different one), then the `workflow://` schema remains consistent for identifying any resource as `workflow://workflow_uuid/task_unique_name/`.

We extended the DagOnStar architecture by including microservices and VCs as building blocks.

B. Microservices as building blocks

In our approach, we encapsulated the applications considered in a workflow into VCs. We added I/O interfaces and control structures to the VCs where the applications have been encapsulated into. This VC represents the first Building Block considered in this approach, which we called *BB-VC*.

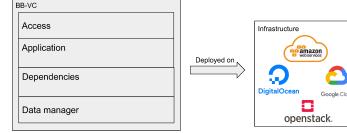


Fig. 2. Conceptual design of a microservice building block in DagOnStar.

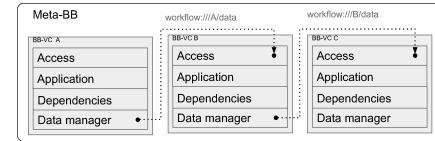


Fig. 3. A microservice (Meta-BB), chaining BB-VC to build a pipeline pattern.

In DagOnStar, the applications are encapsulated into VCs providing the workflow applications with portability property and allowing the deployment of the tasks on multiple platforms. A VC encapsulates the source code or binary of the application, as well as its dependencies libraries, packages, OS, and environment variables and any ancillary component needed to produce the task results. These VCs are constructed in the form of Building Blocks, which we called *BB-VC*.

Figure 2 depicts the conceptual design of a *BB-VC* in the form of a stack. At the top of the stack, there is a BB access control for receiving requests from DagOnStar. It also includes an application layer where a workflow task is placed in the form of source code or a binary. In the next level, are found the dependencies required by an application to execute a task. Finally, on the lower level, a data management component manages the I/O data of the BB-VC.

The *BB-VCs* are grouped in the form of microservices to create a meta Building Block (*Meta-BB*), which also includes I/O interfaces and control structures. The *Meta-BBs* allow to workflows designers to re-use previous tasks created in by the workflow engine [15], [18].

A *Meta-BB* represents a stage of a workflow, whereas the interconnection of multiple *Meta-BBs* produces a workflow, which can be allocated in different computational resources in a distributed manner.

A *Meta-BB* represents a front-end for a stage in a workflow. This means this structure is masking the management of *BB-VCs* deployed on an infrastructure. In order to do this masking procedure, a *Meta-BB* includes a proxy to distribute a load of tasks to the *BB-VCs* and a load balancing mechanism to keep a fair workload distribution. A *Meta-BB* also includes a *DagOnStar executor* to receive tasks assignments and to deliver a notification to the engine. This means that the engine is delegating the control of tasks distribution to the *Meta-BB*, which executes the tasks as a black box; as a result, a *Meta-BB* is an independent piece of software that can be deployed on different types of infrastructure, which improves the portability of the workflows created by DagOnStar.

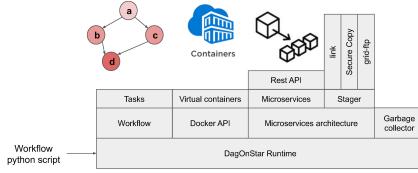


Fig. 4. Deployment of workflows.

The `workflow://` schema previously described is used to chain *Meta-BB*s to build dataflows, which are executed over a microservice ecosystem.

In DagOnStar, a *Meta-BB* has a scratch directory to write their outputs in the form of files (or directories depending on the application outputs) and considering the scratch directory as the root of the local storage supporting the I/O during the task execution. Other *Meta-BB*s and tasks, can receive these files as input, which as to be staged in order to be processed. Files are transported between *Meta-BB* by using link/copy where they are in the same machine, or by using a data transference application/protocol (secure copy, grid-FTP [19], or a content delivery network such as SkyCDS [20]) when they are distributed through different computers.

C. Building parallel Patterns in *Meta-BBs*

In order to improve the efficiency of the workflows, developers can create patterns within a *Meta-BB* for facing up the performance heterogeneity. For instance, Figure 3 shows a conceptual design of the chaining of *BB-VCs* by using `workflow://` scheme to create a pipeline of *BB-VCs*. The microservice is in charge of the management of data. Notice that DagOnStar enables developers to create different types of patterns.

The implementation of a *Meta-BB* exposes an HTTP Rest API, which contains all the functions that allow microservices to communicate with each other. It also includes functions to exchange data and to create dataflows. This means *Meta-BBs* can be chained to produce a workflow modeled as a directed acyclic graph [5] in the very fashion performed in the previous version.

Thanks to the improvements described in this work about the management of *BB-VC* and *Meta-BBs*, the DagOnStar tasks can be deployed in a microservice ecosystem, where these tasks are managed and monitored by the core application life support (DagOnStar Runtime, see Figure 4). Deploying tasks as microservices enables the workflow engine to gain executor independence, efficiency and liability taking advantage of the benefits of this technology, such as, but not limited to, fault-tolerant design, decentralized data management, loosely coupling, indecently deploying, self-contained, and limited scope.

D. Life cycle of microservices building blocks

Figure 5 depicts the life cycle of a microservice building block in DagOnStar, which is integrated by seven stages,



Fig. 5. Life cycle of a microservice task.

which are:

- 1) In the first phase, for each *Meta-BB*, is created a scratch directory in the file system of the machine where it is going to be deployed on.
- 2) In the second phase, a VC image is created per each BB-VC considered in a *Meta-BB*. Each image includes the source code of the task and its dependencies on its corresponding BB-VC. In the case of the base image does not exist; it is downloaded from Docker Hub¹ or any other Docker registry configured by the user².
- 3) In this phase, each BB-VC image constructed in the previous phase is linked to the scratch directory with a volume inside the container of this BB-VC. The ports of all BB-VCs are published to be reachable by other microservices and applications.
- 4) In this phase, the *Meta-BB* is published in the DagOnStar service by using a unique token, which is used as an ID in the controlling of accesses to the data managed by a microservice. This allows a *Meta-BB* to receive petitions from other *Meta-BB* in the ecosystem.
- 5) In operation time, the BB-VCs included in a microservice are executed as a DagOnStar task. This means it receives input data, processes these data, and writes results as a file through the output interface.
- 6) The scratch directory is removed when the execution of the tasks has been completed.
- 7) The *Meta-BB* publication is removed from the ecosystem, removing the BB-VCs associated with that *Meta-BB*.

As can be seen, DagOnStar Stager manages the data between workflows and tasks, which reduces the copying of data produced by one task and used by another.

IV. PROTOTYPE IMPLEMENTATION DETAILS

The prototype based on the microservice-based building block approach presented in this paper was implemented mainly in Python programming language using libraries for the management of cloud and Docker tasks. Microservices API was implemented using a Python library called Flask, and are deployed in production by using uWSGI and Nginx as the webserver. Microservices were encapsulated into Docker VCs and deployed on the Docker swarm platform.

V. EXPERIMENTAL EVALUATION METHODOLOGY

An experimental methodology based on two case studies was conducted to evaluate the performance of the prototype described in the previous section.

¹<https://hub.docker.com/>

²<https://docs.docker.com/registry/>

TABLE I
CHARACTERISTICS OF THE MACHINES USED TO CONDUCT THE EXPERIMENTAL EVALUATION.

Host	Sockets	Cores	Threads	RAM	Storage
compute1	2	6	2	64	3.2T
compute2	1	6	1	24	465.8G, 931.5G, 931.5G
compute3	2	8	1	64	931.5G, 931.5G, 931.5G, 931.5G

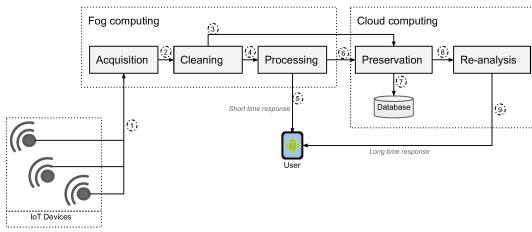


Fig. 6. Workflow for the processing of IoT data.

Two real-world workflow solutions were implemented by using this prototype to conduct two case studies. The first study is based on information for the preparation of IoT data [21]. The second study was based on the processing of satellite imagery captured by the satellite Landsat 8. We implemented a workflow composed of two main microservices: the pre-processing stage and the processing stage.

The prototype was evaluated in the experimental evaluation conducted by using a test environment, including three physical machines. Table I shows the main characteristics of the machines used in the experimental evaluation.

VI. CASE STUDY I: IOT DATA MANAGEMENT

We developed a workflow for collecting and processing environmental data acquired from IoT devices. We deployed a set of *Meta-BBs* to attend IoT devices in the fog computing, whereas another set was deployed on the cloud to process and manage the acquired data.

The data collected from sensors include device temperature, ambient temperature, humidity, motion, light, and Received Signal Strength Indicator.

The dataset was processed through the workflow depicted in Figure 6 which includes the next tasks deployed as *Meta-BB*:

- *Acquisition*: This microservice is in charge of receive the data captured by IoT devices [22], such as instruments/sensors [23] and smart devices. Each device is registered in this microservice, which enables it to send data to the workflow.
- *Cleaning*: This microservice includes the pre-processing of the data collected, removing outliers, and filling missing values by using Z-score technique [24].
- *Processing*: This microservice process the data to identify unusual behavior with these data (i.e., a fire where the sensor has been allocated in, or the dramatic ascend/descend in temperature caused by anomalies in the environment monitored by the sensor. This microservice produces short time alerts with the most recent data

collected by the sensor. These alerts are sent to the application/user indicated in the initial configuration.

- *Preservation*: The microservices were deployed on the cloud to perform the management and preservation of the data collected by the sensors, the cleaned of data and the results of the data processing as well as the logs of the alerts.
- *Re-analysis*: This microservice performs a re-analysis of the data to get an overall summary of the data, performing tasks such as data regression, correlations, and data clustering.

The first three microservices (acquisition, cleaning, and processing) are deployed in fog computing, whereas the preservation and re-analysis are performed in the cloud.

A. Experiments

The workflow constructed in this case study was evaluated in three phases. In the first one, we evaluated the costs of the deployment of *Meta-BBs* in a workflow. The experiments of this phase were performed by measuring the average response time for the construction and deployment of *Meta-BB* images. In the second phase, the deployment of the workflow constructed with *Meta-BBs* was evaluated. The experiments of this phase include the deploying of the studied workflow on three different machines (see Table I), and varying the number of *Meta-BBs* running in parallel for each task in the workflow (see Figure 6). Finally, the performance of the workflow depicted in Figure 6 was evaluated by deploying the workflow with Batch tasks and *Meta-BB* on the infrastructure. It was evaluated the service time spent by this workflow to attend different numbers of concurrent requests sent to the workflow.

B. Results

In this section, are presented and analyzed the results of the performed experiments in the three phases previously described.

- 1) *Analyzing the costs of construction and deployment of building blocks*: Figure 7 shows, in the vertical axis, the response time for the deployment of the VC images (BB-VCs) used to execute each task of the workflow (horizontal axis) showed in Figure 6. For each stage of this workflow, Figure 7 shows the response time in two columns: The first one representing the response time when the BB-VCs are created and the second one when the BB-VCs are reused from images previously created. Each column includes three service times: the first one is the building of the basis image (includes an operating system, programming language compiler, or any other tool necessary to run the program). The second one represents the time required by DagOnStar to prepare the image by adding control structures and I/O libraries to the BB-VC. The last one represents the time to encapsulate an application into a BB-VC by adding the application (either binary or source code) and dependencies on the container. The sum of these times represents the costs of building a BB-VC. This means the first column represents the costs of

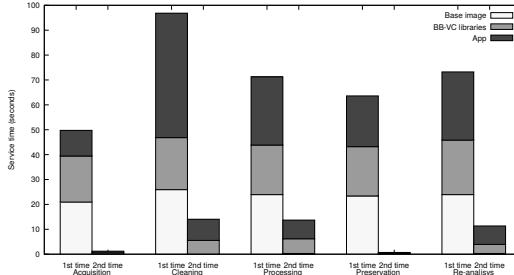


Fig. 7. Average response time for the deployment of tasks in the workflow.

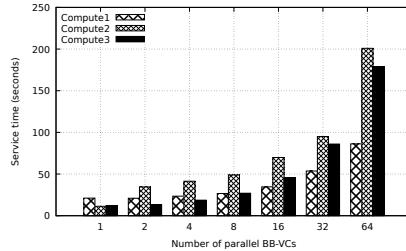


Fig. 8. Average Response time for the deployment of the workflow with different number of parallel BB-VCs in different machines.

building a microservice from scratch, whereas the second one represents the re-utilization of a microservice. As expected, the initiation of a microservice is more expensive than re-using an already configured one (the costs depend on the volume of the applications in both cases).

2) Analyzing the costs of deployment of workflows using Meta-BBs: The results of the second evaluation phase are described in this section. In this phase, we evaluated the costs of deploying a workflow on three different machines (see Table I). This workflow was configured to launch different numbers of BB-VCs in parallel for each Meta-BB (stage) in the workflow. The impact of the number of BB-VCs on the deployment of the workflow was assessed in these experiments.

Figure 8 shows, in the vertical axis, the response time produced by the deployment of the evaluated workflow by using n BB-VCs in a parallel pattern called manager/worker (horizontal axis) for three different machines. As can be seen, the Meta-BBs of DagOnStar transparently manages the parallel patterns by changing the parameter of the number of workers (n). Moreover, the workflows could be deployed on different types of servers without IT staff performing a troubleshooting process. As expected, the more BB-VCs deployed on the infrastructure, the more response time observed by end-users to deploy the whole components of the studied workflow. Please note that the highest cost is associated with the building of the BB-VC basis image as the rest of the BB-VCs are clones, and the costs of these structures are not high. For

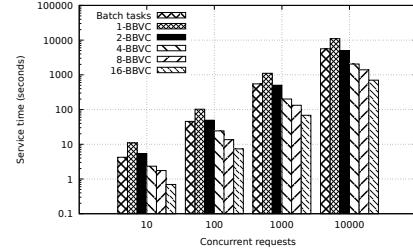


Fig. 9. Average service time for serving different number of concurrent task using different number of services in parallel per performed task.

instance, the deployment of a 64 BB-VC system was deployed on infrastructure in about three minutes.

C. Analyzing the performance of workflows built with Meta-BB

The results of the third phase are described in this section. In this phase, we evaluated the next configurations of the workflow:

- **Batch tasks:** This configuration represents the implementation of the workflow using the regular Batch tasks of DagOnStar.
- **1-BBVC:** This configuration represents the implementation of the workflow by using microservices building blocks implemented in DagOnStar for each stage.
- **n -BBVC:** This configuration represents the implementation of BBVC but deploying parallel patterns for each stage, which improves the service time of processing the workload received by each stage.

To perform these experiments, we configure a bot to create and send artificial requests to the above-described configurations, which accepted and processed these requests as valid requests sent by real end-users. A request represents data sent by an IoT sensor to the workflow to be processed. Therefore, n concurrent requests represent n sensors transmitting data to the workflow (the size of data is homogeneous for all the experiments).

Figure 9 shows on its vertical axis, the service time produced by each configuration of the workflow for attending a varying of the number of concurrent requests (horizontal axis). The results show that Batch tasks produce the best performance comparing with 1 Meta-BB configuration. The response time of *Batch tasks* for attending 10000 concurrent requests was of 93.51 minutes, and 183.5 minutes for 1 Meta-BB. The performance difference of these configurations represents the overhead produced by the microservice managing BB-VCs. Nevertheless, when comparing the *Batch tasks* with the *three Meta-BBs managing 16 BB-VC*, we can see that 16-BBVC configurations produced a percentage of performance gain 87% for an acceleration of 8x as this configuration processed the workload of 10000 concurrent requests in 11.42 minutes.

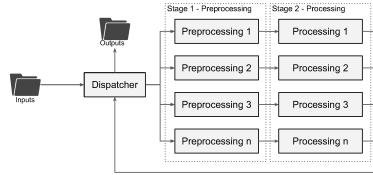


Fig. 10. Workflow for the processing of satellite imagery.

VII. CASE STUDY II: SATELLITE IMAGERY PROCESSING

This case study is based on the processing of satellite imagery captured by the satellite Landsat 8. We implemented a workflow composed of two main microservices(Meta-BB): the pre-processing stage and the processing stage.

A summary of each microservice designed is as follows:

- 1) *Preprocessing*: The pre-processing microservice calculates the radiation and top atmosphere reflectance (TOA) of each of the eleven bands of Landsat 8 images [25], [26].
- 2) *Processing*: The processing microservices include the calculation of the traditional indexes (e.g., NDVI, EVI, SAVI, MSAVI, NDMI, NBR, NBR2 and NDSI [25], [26]).

Figure 10 depicts the conceptual design of the workflow for the pre-processing and processing of the satellite imagery. The workflow was designed in the form of a Manager/Worker pattern. In this pattern, an entity called Manager is in charge of distributing, in a load-balanced manner, a set of images through different workers. The workers execute the two stages of the workflow (pre-processing and processing) and send the results back to the Manager.

A. Results

To test the performance of the pattern described in Figure 7, we used a repository of 20 images (of a size of 17.8 GB) captured by LandSat8.

Figure 11 shows in the vertical axis, the response time for the execution of the whole workflow created by using a different number of parallel BB-VCs (horizontal axis).

As it can be observed in Figure 11, the response time is reduced for each workflow when increasing the number of BBVC running in parallel. For instance, the pattern running 8 BBVC produces a response time of 3.83 minutes, whereas the pattern running with only one BBVC produces a response time of 25.03 minutes. This means an improvement in the response time of 84.69% comparing the pattern with eight parallel BBVC with the pattern with only one BBVC.

B. Performance comparison of the approach with state of art solutions

We performed a direct comparison of the proposed approach based on Meta-BBs and BB-VC with other two workflows engines in the literature: Parsl [8] and Makeflow [8]. The workflow used for testing was the one depicted in Figure 7 for the pre-processing and processing of satellite imagery.

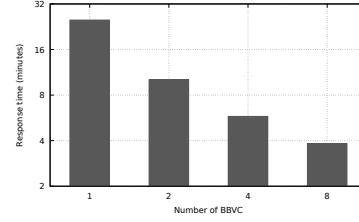


Fig. 11. Response time for the execution of the preprocessing and processing pattern.

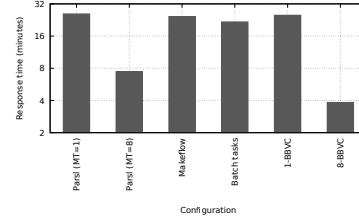


Fig. 12. Response time of the workflow implemented in different workflow engines.

The workflow executed with Makeflow was configured to run over virtual containers (parallel patterns are not available in this solution). For our approach, we tested two configurations of the workflow with one and eight BB-VC in the patterns inside of the Meta-BBs. We also configured Parsl to use two different configurations: the first one using one single thread and the second one using eight threads. We called this configurations as *Parsl*($MT = 1$) and *Parsl*($MT = 8$) respectively.

Figure 12 shows the response time (vertical axis) produced by the studied solutions for the processing of the 20 images.

As it can be seen in Figure 12, the approach using 8-BBVC configuration produced the best performance compared with both configurations of Parsl and the solution developed with Makeflow. 8-BBVC produced a response time of 3.83 minutes, whereas *Parsl*($MT = 8$) produced a response time of 7.43 minutes and *Makeflow* produced a response time of 24.34 minutes, which means that 8-BBVC yielded an improvement in the response time of 48.42% and 84.25% in comparison with *Parsl*($MT = 8$) and *Makeflow* respectively.

TABLE II
QUALITATIVE COMPARISON OF DAGONSTAR+MICROSERVICES

Engine	Tasks Portability	Tasks deployment		Tasks re-usability	Data decoupled
		Fully automatic	Semi automatic		
Galaxy [1]	Partially		✓		
Parsl [2]	Partially		✓		
Makeflow [8]	✓		✓	✓	
DagOnStar [5]	✓		✓	✓	
DagOnStar + Microservices	✓	✓		✓	✓

VIII. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we presented a novel contribution in DagOnStar development, enabling the lightweight workflow engine to support the microservice technology in a VC context. As previously described in [5], DagOnStar has been designed with simplicity in mind focusing on production scenarios in the field of massive data computational environmental science applications such as extreme weather predictions pollution forecasting [27], crowdsourced environmental data processing [28] and distributed weather stations network data analysis [21].

The evaluation revealed the feasibility of using a building block approach to automatic deploy workflows in a microservice environment. It also revealed the flexibility of this approach to improve the performance of the workflows created by chaining *Meta-BB* in the form of parallelism patterns.

From the task typology point of view, we are going to integrate the DagOnStar microservice-based task interaction with GPU intensive computing tasks in virtualized [29] and containerized architectures even leveraging on mobile and single-board computing devices [30].

We plan to evolve DagOnStar in order to provide application live support to instrumented data acquisition platforms as vehicles, vessels, and drones [31] scenarios where the data processing workflow throughput is crucial because of the mission critical-like production conditions.

ACKNOWLEDGMENTS

The research project supported this research “DYNAMO: Distributed leisure Yacht-carried sensor-Network for Atmosphere and Marine data crowdsourcing applications” (DSTE373) and it is partially included in the framework of the project “MQAP - Maritime Operation Quality Assurance Platform” and financed by Italian Ministry of Economic Development.

REFERENCES

- [1] R. Madduri, K. Chard, R. Chard, L. Lacinski, A. Rodriguez, D. Sulakhe, D. Kelly, U. Dave, and I. Foster. The globus galaxies platform: delivering science gateways as a service. *Concurrency and Computation: Practice and Experience*, 27(16):4344–4360, 2015.
- [2] Y. Babuji, A. Woodard, Z. Li, D. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, et al. Parsl: Pervasive parallel programming in python. In *28th HPDC*, pages 25–36. ACM, 2019.
- [3] T. McPhillips, S. Bowers, D. Zinn, and B. Ludischer. Scientific workflow design for mere mortals. *FGCS*, 25(5):541–551, 2009.
- [4] M. P Singh and M. Vouk. Scientific workflows: scientific computing meets transactional workflows. In *NSF Workshop*, pages 28–34, 1996.
- [5] R. Montella, D. Di Luccio, and S. Kosta. Dagon*: Executing direct acyclic graphs as parallel jobs on anything. In *WORKS 2018*, pages 64–73. IEEE, 2018.
- [6] I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer, 2007.
- [7] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
- [8] M. Albrecht, P. Donnelly, and D. Bui, Peter P. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on SWEET*, page 1. ACM, 2012.
- [9] M. Wilde, M. Hategan, J. M Wozniak, B. Clifford, D. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.
- [10] C. Zheng and D. Thain. Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *8th VTDC*, pages 31–38. ACM, 2015.
- [11] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D’Souza, S. Devloid, D. Murphy-Olson, N. Desai, et al. Skyport: container-based execution environment management for multi-cloud scientific workflows. In *5th DataCloud*, pages 25–32. IEEE Press, 2014.
- [12] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In *SSDBM*, pages 471–481. Springer, 2010.
- [13] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain’t markup language (yaml) version 1.1. *yaml.org, Tech. Rep.*, page 23, 2005.
- [14] N. Dragoni, S. Giallorenzo, A. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.
- [15] N. T. de Sousa, W. Hasselbring, T. Weber, and D. Kranzmüller. Designing a generic research data infrastructure architecture with continuous software engineering. In *CSE 2018*. CEUR-WS. org, 2018.
- [16] Yong Zhao, Mihai Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stefan-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *2007 IEEE Congress on Services (Services 2007)*, pages 199–206. IEEE, 2007.
- [17] R. Montella, D. Kelly, W. Xiong, A. Brizius, J. Elliott, R. Madduri, K. Maheshwari, C. Porter, P. Vilner, M. Wilde, et al. Face-it: A science gateway for food security research. *Concurrency and Computation: Practice and Experience*, 27(16):4423–4436, 2015.
- [18] D. Sánchez-Gallegos, JL Gonzalez-Compean, S. Alvarado-Barrientos, Victor J Sosa-Sosa, J. Tuxpan-Vargas, and J. Carrero. A containerized service for clustering and categorization of weather records in the cloud. In *2018 CSIT*, pages 26–31. IEEE, 2018.
- [19] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *2005 ACM/IEEE Supercomputing*, page 54. IEEE Computer Society, 2005.
- [20] J. L. Gonzalez, J. C. Perez, Vor J Sosa-Sosa, Luis M Sanchez, and B. Bergua. Skycds: A resilient content delivery service based on diversified cloud storage. *Simulation Modelling Practice and Theory*, 54:64–85, 2015.
- [21] D. Sánchez-Gallegos, D. Di Luccio, JL. Gonzalez-Compean, and R. Montella. Internet of things orchestration using dagon* workflow engine. In *2019 WF-IoT*, pages 95–100. IEEE, 2019.
- [22] G. Laccetti, R. Montella, C. Palmieri, and V. Pelliccia. The high performance internet of things: using gvirtus to share high-end gpus with arm based cluster computing nodes. In *PPAM*, pages 734–744. Springer, 2013.
- [23] R. Montella, G. Agrillo, D. Mastrangelo, and M. Menna. A globus toolkit 4 based instrument service for environmental data acquisition and distribution. In *3rd Upgrade-cn*, pages 21–28. ACM, 2008.
- [24] Dhiren Ghosh and Andrew Vogt. Outliers: An evaluation of methodologies. In *Joint statistical meetings*, pages 3455–3460, 2012.
- [25] Gyanesh C., Brian L. M., and Dennis L. H. Summary of current radiometric calibration coefficients for landsat mss, tm, etm+, and eo-1 ali sensors. *Remote Sensing of Environment*, 113(5):893 – 903, 2009.
- [26] Alexander Ariza. Descripción y corrección de productos landsat 8 ldcm (landsat data continuity mission), 2013.
- [27] R. Montella, D. Di Luccio, P. Troiano, A. Riccio, A. Brizius, and I. Foster. Wacomm: A parallel water quality community model for pollutant transport and dispersion operational predictions. In *2016 SITIS*, pages 717–724. IEEE, 2016.
- [28] R. Montella, D. Di Luccio, L. Marcellino, A. Galletti, S. Kosta, G. Giunta, and I. Foster. Workflow-based automatic processing for internet of floating things crowdsourced data. *FGCS*, 94:103–119, 2019.
- [29] R. Montella, S. Kosta, D. Oro, J. Vera, C. Fernández, C. Palmieri, D. Di Luccio, G. Giunta, M. Lapeagna, and G. Laccetti. Accelerating linux and android applications on low-power devices through remote gppu offloading. *Concurrency and Computation: Practice and Experience*, 29(24):e4286, 2017.

- [30] R. Montella, C. Ferraro, S. Kosta, V. Pelliccia, and G. Giunta. Enabling android-based devices to high-end gpgpus. In *ICA3PP*, pages 118–125. Springer, 2016.
- [31] R. Montella, S. Kosta, and I. Foster. Dynamo: Distributed leisure yacht-carried sensor-network for atmosphere and marine data crowdsourcing applications. In *2018 IC2E*, pages 333–339. IEEE, 2018.

An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience.

Dante D. Sánchez-Gallegos^a, Diana Di Luccio^b, Sokol Kosta^c, J. L. Gonzalez-Compean^a, Raffaele Montella^b

^a*Cinvestav Tamaulipas, Cd. Victoria, Mexico*

^b*Department of Science and Technologies - University of Naples “Parthenope”, Naples, Italy*

^c*Department of Electronic Systems - Aalborg University Copenhagen, Denmark*

Abstract

Workflow engines are commonly used to orchestrate large-scale scientific computations such as, but not limited to weather, climate, natural disasters, food safety, and territorial management. However, to implement, manage, and execute real-world scientific applications in the form of workflows on multiple infrastructures (servers, clusters, cloud) remains a challenge. In this paper, we present DagOnStar (*Directed Acyclic Graph On Anything*), a lightweight Python library implementing a workflow paradigm based on parallel patterns that can be executed on any combination of local machines, on-premise high performance computing clusters, containers, and cloud-based virtual infrastructures. DagOnStar is designed to minimize data movement to reduce the application storage footprint. A case study based on a real-world application is explored to illustrate the use of this novel workflow engine: a containerized weather data collection application deployed on multiple infrastructures. An experimental comparison with other state-of-the-art workflow engines shows that DagOnStar can run workflows on multiple types of infrastructure with an improvement of 50.19% in run time when using a parallel pattern with eight task-level workers.

Keywords: Workflow, Data Intensive, Cloud Computing, Directed Acyclic Graph, Parallel Processing

1. Introduction

Workflow engines are widely used to conduct large-scale studies (e.g., weather [1], climate [2], natural disasters [3], food safety [4, 5], and territorial management [6]) in an efficient manner [7, 8]. These processing structures enable developers to couple distributed heterogeneous applications with services on large scale infrastructures [9, 10, 11]. Workflow engines, which can be used for data-intensive science, have existed since the beginning of the grid computing era [12]. Examples of these engines include Triana [13], Taverna [14], Kepler [15], Unicore [16], and Pegasus [17].

In recent years, different ambitious projects have based their solutions on workflows [18, 19, 20]. All such solutions have faced one common main challenge, which was to simplify the interaction of domain scientists with computational resources [21]. This comes as a consequence that it was assumed that infrastructure would be available at any moment and enough to support workflows [22, 23].

Cloud-based solutions have continued to provide elasticity and virtually infinite computational resources to workflow designers opening intriguing scenarios involving GPUs, IoT and acceleration at the edge [24]. Also, these cloud solutions have been an opportunity for organizations to reduce costs and to outsource complex IT services and infrastructure management [25, 23]. High-end computing infrastructures are playing a primary role in computational sciences, as the infrastructure required for organizations is online and virtually available on-demand in an immediate manner in the cloud [26].

Nevertheless, new challenges arise when organizations deploy workflows on this type of pervasive and ubiquitous services [27] because outsourcing computing and storage services implies also outsourcing control over those services. For example, issues such as outages (reliability) [28], violations of access controls [29] (confidentiality), and the feasibility to move solutions to different providers (vendor lock-in) [27, 30] can arise. Today, workflows are first-class citizens in the distributed computation ecosystem performing strategic tasks (data analysis, machine learning, loosely coupled parallel computation), especially in IoT scenarios where a large amount of data is continuously produced [31]. This can lead to performance and affordability issues in dynamic ubiquity outsourcing scenarios based on the pay-as-you-go model.

In this context, there is a need for flexible tools enabling organizations to develop and deploy workflows on different types of infrastructures. This

involves enforcing properties such as portability, profitability, and efficiency to prevent solutions from suffering either from vendor lock-in side effects in migration scenarios or from troubleshooting when trying to reproduce experiments in different infrastructures. For example, Zhao et al. [32, 33] examined 92 workflows submitted to Taverna from 2007 to 2012, and found that 80% either did not execute or did not produce reported results.

The automatic deployment of workflows on different infrastructures remains a challenge [34], even though data preparation [35], multi-cloud [36], and virtual container clusters [37] have been proposed for outsourced services and big data, and workflow engines such as Parsl [38], Pegasus [39], and Makeflow [40] are considering portability based on containerized solutions.

In this paper, we describe DagOnStar¹ (*Directed Acyclic Graph On Anything*), an open source Python library (and related components) enabling the execution of directed acyclic graph (DAG) jobs on diverse computational resources, ranging from the local machine to virtual HPC clusters hosted on private, public, or hybrid clouds [41].

According to the requirements needed by environmental science applications using workflows [42], and adopting an application-targeted design approach [43], the desired features in such a tool are the following:

- Integration in the Python programming environment;
- Minimal footprint in terms of storage room used for the external software components' execution sandbox;
- Avoid any workflow engine centered management of data (need to share the file system between computational nodes);
- Easy definition of tasks with directed use of Python scripts, web interaction, external software components execution, cloud-hosted virtual machines, or containerized tools;
- Enable execution of tasks on heterogeneous computing resources.

These requirements drove the DagOnStar design *ab-initio*, targeting the computational scientists' needs since its first prototype. Moreover, the de-

¹Dagon was a Phoenician god, known by the ancient Greeks as Triton.

velopment of DagOnStar has been motivated by our research involving operational application for routinely produced weather and marine forecasts for the Center for Monitoring and Modelling Marine and Atmosphere Applications (CMMMA) operated by the Department of Science and Technologies at the University of Naples Parthenope² and the automatic acquisition, pre-processing, and analysis of the Mexican weather stations network (EMAS), covering the whole Mexican territories. DagOnStar is characterized by several peculiar features making it different from similar software components already available. These features include: (i) the use of the `workflow://` schema for implicit data-flow definition and sandbox based execution environment management, (ii) the use of parallel patterns in order to enforce task-level parallelism leveraging manager/workers paradigm; (iii) the orchestration of sensors and actuators available within the IoT paradigm [37] alongside regular computing-intensive tasks, and (iv) the minimization of the data movement footprint.

The specific contributions of this paper include:

- A pattern-based model for applying implicit parallelism to the tasks required in creating data-flows in workflows through shared storage footprints.
- An Internet of Things workflow task model enabling the computation of bulk synchronous parallel jobs close to the data acquisition [44]. DagOnStar tasks can run in *continuous mode*: in which the task execution is resubmitted each time the task processing ends in order to be ready to process new incoming data [45]. The proposed approach implements cyclic behavior at task level without breaking the DAG paradigm.
- A containerizing scheme for adapting to the deployment of workflows, in an efficient and flexible manner, which can be applied on multiple scenarios (e.g., processing of IoT data) where tasks can be deployed on any edge, fog, cloud, or end-user device. In this scheme, the tasks are encapsulated into virtual containers with control structures that implicitly and transparently manage parallelism, load balancing, and its execution as well as the management, delivery, and retrieval of data.

²<http://meteo.uniparthenope.it>

This avoids the requirements of installing third-party solutions for managing resources and parallelism as well as to fine-tune parameters for finding efficient configurations for each case.

The rest of this paper is as follows: Section 2 describes the related work and presents a discussion about the state of the art and the proposed solution. Section 3 discusses design strategies, parallelism, and the distribution approach of DagOnStar. Section 4 describes the DagOnStar application lifecycle in detail. Section 5 describes a real-world operational applications that drove and accelerated the development of DagOnStar. Section 6 presents the evaluation performed to test DagOnStar. Finally, Section 7 provides concluding remarks and outlines the path for future research development.

2. Related Work

Workflow engines are traditional solutions enabling end-users to execute a sequence of applications (tasks) in different computational resources (mainly distributed) [46]. These engines automatically inject data into each task in a workflow to create continuous processing through the tasks of the workflow (dataflows) [47]. The engines also consider mechanisms for users to collect and visualize the output data produced by each task in a workflow.

In recent years, these types of processing structures were quite popular in grid computing environments and represented the traditional solution for organizations to support large scale scientific studies [48, 15, 12]. This technique is also used in new environments such as cloud computing and clusters of virtual containers [40, 49, 50].

In this context, workflow engines did not make the transition to the cloud or containerized environments [13, 51] and there is no more support for these solutions. Other workflows still have the support and are mainly deployed on high-performance infrastructure [52, 53, 54, 55]. Teams have started the adaptation of their workflow engines to new computing environments. Some examples of popular engines that have started this transition are Parsl, Makeflow, Pegasus, PyCOMPSs [53], and Globus Galaxy [10]. In this sense, DagOnStar is similar to these workflows. Therefore, in this section, we describe the similarities and differences of these solutions through a qualitative comparison.

Table 1 presents a qualitative comparison of DagOnStar with the other workflow engines introduced above from the perspectives of efficiency, fault-tolerance, continuity in the delivery of data, and scalability.

Table 1: Qualitative comparison of DagOnStar with state of the art workflow engines. \ddagger means that the engine uses Condor or MPI to provide task parallelism, $*$ means that only the virtual container tasks are fully portable and these are not automatically built and \dagger means that the scalability is provided by configuring an external tool such as Condor.

Work	Reliability (fault-tolerance)	Efficiency Patterns	Portability Threads	Continuity Workflow	Scalability Tasks	Security Delivery	Vertical	Horizontal	Site	Transport
Galaxy [10]	✓		✓	✓	✓	✓				✓
Makeflow [40]	✓		✓ \ddagger	✓	✓*	✓	✓ \dagger	✓ \dagger		✓
Pegasus [39]	✓		✓ \ddagger	✓	✓*	✓	✓ \dagger	✓ \dagger		✓
Parsl [56]	✓		✓	✓		✓		✓		✓
DagOnStar	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

For **reliability**, the ability of the engines to withstand the failure of tasks in the workflow was considered. This property is satisfied by all engines considered in this comparison.

The **efficiency** property is measured depending on the way the parallelism is implemented by the engines. As it can be seen, threads per task is a common strategy, which is implemented in some cases (\ddagger in the case of Pegasus [39] and Makeflow [40]) invokes other tools (e.g., Condor [57] or MPI) to provide task parallelism, which implies that these engines require installing additional tools to provide solutions with parallelism.

Based on the Swift model [49], Parsl [56] is a software component leveraging on a Python parallel scripting library, supporting asynchronous and implicitly parallel data-oriented workflows. This means this engine is similar to DagOnStar in the implicit fashion in which tasks are executed in a concurrent manner. Instead of launching threads per each content to process as performed by traditional engines, DagOnStar produces implicit parallelism established in each task of the workflow by using containerized parallel patterns and load balancing algorithms (also used in microservice mesh [58, 59]). This feature enables DagOnStar to establish different parallelism patterns per task. In scenarios of tasks producing heterogeneous performance in a workflow, this feature produces a significant improvement in the workflow performance when processing large data-sets as delays produced by bottlenecks can be reduced by tuning the parallel patterns for these tasks. DagOnStar and Parsl do not require installing third-party tools to execute parallel tasks, which preserve the data dependencies between tasks.

In practice, to deploy a workflow on different infrastructures, it is common that workflow developers perform troubleshooting procedures in the tasks of

Table 2: Containers managements in workflow engines.

Work	Execution infrastructure				Third-party dependencies	VC image acquisition		Encapsulation level		Microservices support	
	VM	VC	Cluster	Single machine		TAR file	Preloaded	Built	Workflow	Task	
Pegasus	✓	✓	✓	✓	Condor, Docker/Singularity	✓			✓	✓	
Makeflow	✓	✓	✓	✓	Condor, Docker	✓	✓		✓		
Parsl	✓	✓	✓	✓	Kubernetes		✓			✓	
DagOnStar	✓	✓	✓	✓	Docker	✓	✓	✓	✓	✓	✓

workflow such as dependencies of the workflow, configuring environment variables, and configuring the apps [60]. These troubleshooting procedures could impact the efficiency and continuity of workflows because of bad configurations in the apps of the workflows. In this sense, virtual containers have become a solution for workflow developers to encapsulate their applications and deploy them through different infrastructures, reducing the time spent in troubleshooting procedures [40]. The **portability** was measured by analyzing the deployment of tasks and the whole workflow. All the engines perform the deployment of workflows on a given infrastructure by using scripts.

Makeflow [40] and Pegasus [39] are workflow engines focused mainly on the portability of workflows and their execution through different IT infrastructures. Pegasus, Makeflow, and DagOnStar thus consider virtual containers to deploy tasks (see Table 1), which, however, are not automatically coupled in the workflow. Table 2 summarizes the management and usage of virtual containers in Pegasus, Makeflow, Parsl, and DagOnStar. All the engines require a third-party tool for the management of virtual containers (e.g., Docker or Singularity), nevertheless, DagOnStar does not require the installation of other third-party solutions to orchestrate the execution of the virtual container, which is the case of the other engines (Condor for Makeflow and Pegasus and Kubernetes for Parsl). In the containerization scheme proposed in DagOnStar, these software pieces are integrated with the tasks executed to create parallel patterns encapsulated into a virtual container. Moreover, DagOnStar implements a containerization scheme including the orchestration of containerized blocks (including parallel patterns) that are automatically coupled in the form of workflows and deployed on different types of infrastructures, which, does not appear to be considered by other engines. Finally, DagOnStar is the only engine that supports the management of tasks as microservices by using the DagOnStar Service [61], which enables the re-utilization of tasks previously deployed, and the management

of the tasks by using a common HTTP API interface.

In cloud computing, **security** is an important property and the engines should add it to a workflow before deploying it on the cloud. As can be seen, traditional engines only prepare data for transportation but not for the preservation of storage. In turn, DagOnStar not only prepares data for transportation but also adds access controls to the data as data are kept encrypted in cloud storage. The usability of these solutions is another important property to take into account by workflow developers.

Galaxy [10] was proposed to perform computational data processing over genomic data with automatic and unobtrusive provenance tracking [62, 63]. To extend Galaxy to support other applications, FACE-IT Globus Galaxy [64, 18] was proposed. The usability of this type of solution is evident as it has been implemented in agricultural and climate modeling, weather/marine forecasts production, and food quality assessment for mussel farms [65]. However, these solutions were designed assuming that some actions associated with the workflow deployment would be performed manually by developers as well as having an available infrastructure to manage the data transfers [66]. The principle designs used to implement these considerations are described in detail in the design section (Section 3).

3. Design of an efficient pattern-based approach

DagOnStar has two main components (see Figure 1): *i*) a *Python library* implementing the application life-cycle at run-time, which is in charge of deploying tasks comprised of a workflow and which delivers the input data to each task in the workflow, and *ii*) a *Service* component that monitors the execution of the workflows during run-time and manages the state of tasks and the workflow, relaunching a workflow if a task fails during its execution.

The DagOnStar Runtime performs the interaction with the actual executors on the premises of local or remote resources, virtual machine instances on public/private/hybrid clouds, and containers.

3.1. The workflow:// schema

The Task component takes charge of the management of data dependencies by using the DagOnStar paradigm designed around the `workflow://` schema, defined as:

`workflow:// unique workflow name/unique task name/resource`

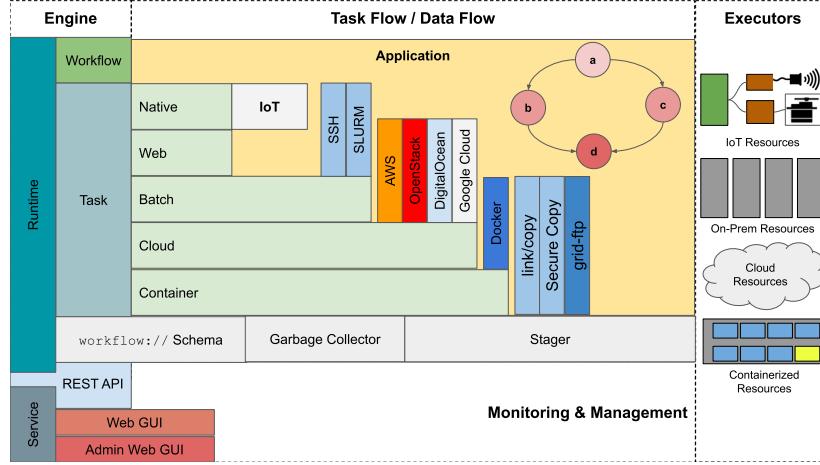


Figure 1: DagOnStar architecture. The lightweight workflow engine is represented on the left side. Diverse and different Task components implement the actual workflow as a task or data flow. The right side of the figure shows four types of executors: IoT components, high-performance computing resources, cloud computing facilities, and containerized environments. The DagOnStar service (bottom part of the figure) is used, optionally, for monitoring, management, and interactions between multiple workflows running on the same or different machines.

where:

- *unique workflow name* is the name of the workflow. If empty, the current workflow is assumed. (Two or more workflows can be created and run in the same DagOnStar application.)
- *unique task name* is the name of the task within the workflow. It must be unique and it cannot be empty. We may consider *unique workflow name/unique task name* as the root of the task scratch directory if a sandbox is used.
- *resource* is the name of the resource produced or managed by the task. The resource can be mapped over input/output Python variables, while with Batch tasks, a resource, is a regular data file.

As in Galaxy, but differently than Parsl, the external software runs in a sandbox implemented as a scratch directory in which a customized, highly

configurable environment is enforced. The shell script wrapping the external software provides the environment configuration, such as the local directory structure, libraries, and ancillary software components, and configuration files.

Input files must be moved to the sandbox execution environment to be processed. The outputs are produced in the form of files (or directories) considering the scratch directory as the root. Usually, at the end of the process the results are moved out of the sandbox.

The `workflow://` schema is used for both the automatically resolving data-flow task dependencies and the garbage collector management. It can be used in conjunction with the DagOnStar Service to receive web-socket based notifications.

3.2. DagOnStar Tasks

A DagOnStar application is developed as a Python script using any Python extension with a regular sequential approach. The `Task` component represents any of the applications, functions, and binary or source codes that perform a processing task in a workflow.

The `Task` component is responsible for managing the input data dependencies of the processing of these data, and the staging of the output data to other tasks. This component also participates in the monitoring process sending the state of the task to the DagOnStar Service.

DagOnStar tasks are independent of infrastructure and the underlying platform, allowing for the implementation of more specialized tasks as an extension of the generic “Task” component (see Figure 2). This characteristic allows DagOnStar to manage different types of tasks, providing the deployment of tasks in any of the local environments (i.e., as a local or remote pure Python component or as an external piece of software).

Tasks can run in *continuous* mode: once the computation is performed and completed, the task returns to the waiting state for new data from its parent-related tasks. All tasks can be set as continuous, performing a never-ending cyclic behavior, meanwhile enforcing the DAG pattern. The continuous mode has been designed to support the IoT workflows where data has to be processed close to the acquisition to minimize or optimize the data movement between distributed IoT nodes or perform information extraction and data reduction in fog/mist computing models [67].

The different task types are classified as follows:

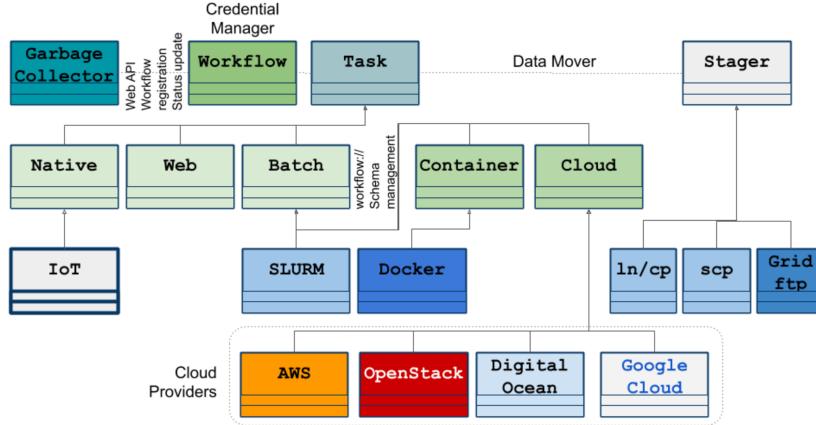


Figure 2: DagOnStar class diagram. All task specializations (in light green) are inherited from the Task class. The Batch class is extended to provide features related to the local job schedulers (in light blue), to the containerized task execution (both locally hosted or cloud-hosted, in blue), and the cloud virtual machine management (in cloud providers-related colors). Finally, the Stager class encapsulates the data movement between task executors strategy. The color scheme is consistent with Figure 1.

- **Native.** Regular Python functions, which are executed locally in a concurrent application thread. This category of tasks has been designed to allow for the implementation of lightweight operations that are conveniently executed in parallel with no need for a specific computing power or storage capacity. The Native tasks are fully integrated with the Python programming environment and are used as base components for tasks that require interaction with machine native libraries, including but not limited to, the IoT Tasks. This type of task shares the same object scope as the DagOnStar application. If executed remotely, the I/O serialization is automatically performed.

- **IoT.** The IoT Task component is built on functional modules:

Interface. It consists of two components working in a specular fashion. The Input Adapter maps a file in a Python variable which is accessible by the IoT Task implementation (for example *workflow://rpi1/bme208* refers to an environmental sensor managed by

a task called `rpi1` running on a Raspberry Pi). The Output Adapter works in a similar fashion, but for actuators instead of sensors. This approach ensures that the `worfkflow://` schema works even in the IoT Task context.

Proxy. It is defined as an interface designed after the typical Arduino-like programming model style. The IoT Task is implemented defining the callback functions for sensors and actuator definition and initialization, the main loop kernel, and what must be performed when the exit condition is reached.

Abstractions. Communication with the actual IoT device can be implemented through direct access to the low-power hardware capabilities, GPIOs, and serial ports or by using the cloud/IoT management framework as AWS-IoT or Stack4Things.

Communicator. A modular component depending on the Abstractions since it is related to the technology used to perform the communication between the machine executing the DagOnStar IoT Task and the IoT Device. Different types of communicators working alongside the DagOnStar IoT Task are based on WebSocket and REST APIs.

IoT Device. IoT Devices with no restrictions concerning computational, storage, GPIO, and communication capabilities. Devices such as Texas Instruments Simplelink CC1350 SensorTag, UDOO single board computers, Raspberry Pi, Arduino Yun, and ESP32 were successfully tested under the development of the IoT Task.

- **Web.** Web tasks have been designed to allow the developer to enable workflows interacting with remote resources accessible using web services. This task type consumes a SOAP web service and invokes the API REST. It waits for an external event and interacts with other DagOnStar workflows leveraging on the DagOnStar Service. Two or more DagOnStar workflow apps can interact with each other using the web services produced by the DagOnStar Service and this kind of task.
- **Batch.** These are external software components executed by using SSH or a local scheduler. This kind of task component, alongside the Cloud and the Container task types, is the base interface for external executors. While Native and Web Tasks are executed within the

DagOnStar application as a local or remote Python component, Batch represents an external software component that can be executed on the same machine, on another machine in the same cluster, or on any remote machine of which the user has been granted permissions to access and perform program execution.

The external software is executed in a scratch directory acting as a sandbox for a customized running environment managed by the DagOnStar garbage collector (see Section 3.5). The data dependencies are defined using the DagOnStar `workflow://` schema (see Section 3.1). The DagOnStar design takes the benefit of the object-oriented paradigm defining a plug-in architecture for the actual batch executors and it has an impact on the workflow programming model (Section 4.1).

- **Cloud.** A task can be embedded in a virtual machine image and executed on the cloud. Cloud tasks leverage on cloud-specific libraries such as Boto for Amazon Web Services³ or cloud-generic components such as Apache Libcloud⁴, using a cloud interface meta-model paradigm [68].

When this kind of task is used, a virtual machine image must be prepared and stored in the cloud image repository. The image must provide life support to the external software with the needed configuration for the stage operations. The virtual machine image can be prepared with the support of a shared file system, Globus transfer [66], SkyCDS [36], or secure copy.

At the time of executing the task, a new virtual machine instance is created using the previously prepared virtual machine image. When the instance is up and running, a local scratch directory is created and data are staged in. Once data is correctly staged in the remote scratch directory, the actual external software is launched. The outputs can be explicitly staged out to a well-known data sink or remain in the scratch directory until they are needed at a later time.

The DagOnStar garbage collector (see Section 3.5) takes charge of the virtual machine stop and/or terminate management.

- **Container.** A task can be represented by a container script and ex-

³<https://aws.amazon.com/it/sdk-for-python/>

⁴<https://libcloud.apache.org>

ecuted on a *containerized* infrastructure. Lightweight container technology has recently risen as an alternative to complete virtualization, saving consistent CPU and input/output costs [40]. This type of task is one of the most ambitious features of DagOnStar. While its specifications have been formally defined, the actual implementation is limited to a naive approach working with Docker scripts [69] wrapping each task at the workflow level and running each task inside a container. This approach limits the scope of the container while it has consistent costs for the startup and shutdown operations.

When a containerized task is executed, the container manager takes charge of the container description script in which the features needed to bind the external software component, the data transfer mechanism (i.e., shared file system, grid-ftp, or secure copy), the creation of the scratch directory, and the external software component itself have been previously coded. Once the container is ready, the task in process begins, and when it is completed and the external software is launched. As already described for the cloud task type, the outputs can be explicitly staged out to a well-known data sink or remain in the scratch directory until they are still needed. The DagOnStar garbage collector (Section 3.5) takes charge of the container shutdown management.

The DagOnStar application developer can extend any **Task** component to design the app in terms of tools providing to the final user a more component-oriented approach. The **Workflow** component is the container for the tasks. The relations between tasks can be defined explicitly (task flow) or by using the DagOnStar data dependence model (data flow) based on the `workflow://` schema. This component is in charge of the evaluation of all the dependencies between tasks and manages the data movement from one task to another when those dependencies involve data.

The Workflow component performs the initial registration and the final de-registration of the managed workflow on the DagOnStar Service. The same DagOnStar application can be built using one or more Workflow instances. Each one is uniquely identified by a 128 bit universally unique identifier. This approach allows for multiple workflows belonging to the same or different DagOnStar application to interact with each other via the DagOnStar Service and the Web tasks. The Workflow component has been designed to manage a checkpoint/resume feature.

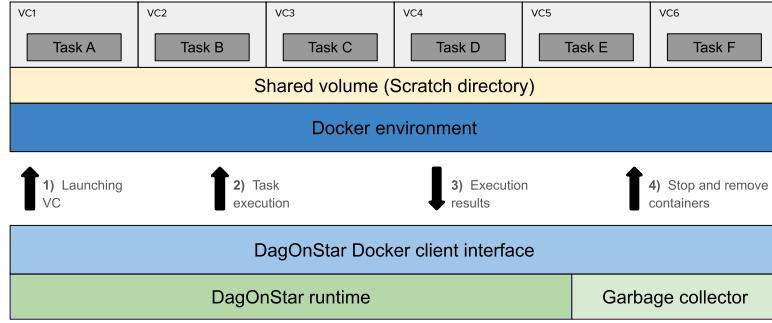


Figure 3: Conceptual design of the management of VC in DagOnStar.

The **Stager** component, deputed to automatically manage the stage-in and stage-out operations, is not directly used by the DagOnStar application developer, but it is managed by the Workflow component to mitigate the need to physically copy data produced by one task and used by another. At the time of performing the actual data stage-in, this component automatically selects the data transfer model concerning the kind of task. This happens when all the dependencies are resolved and just before the external software component must be run.

The Stager component performs a symbolic data link to avoid useless explicit data copying whether the external software component runs on the local machine or a remote machine sharing the same file system. This approach appears to simplify the overall workflow data management.

The Stager component can select different data transport applications implemented in DagOnStar: secure copy (SCP) [70], Globus [66], and SkyCDS [36]. It can select the Globus/ or SkyCDS copy if available, otherwise, the regular secure copy of the external software component must be executed on a remote machine, a virtual machine instanced in a public, private or hybrid cloud, or a container running in local or dedicated infrastructure.

3.3. Deploying tasks on virtual containers

Tasks in DagOnStar can be deployed in virtual containers (VC) using the Docker platform. Furthermore, an interface was designed to send commands to the Docker environment to manage the containers, for example, to launch

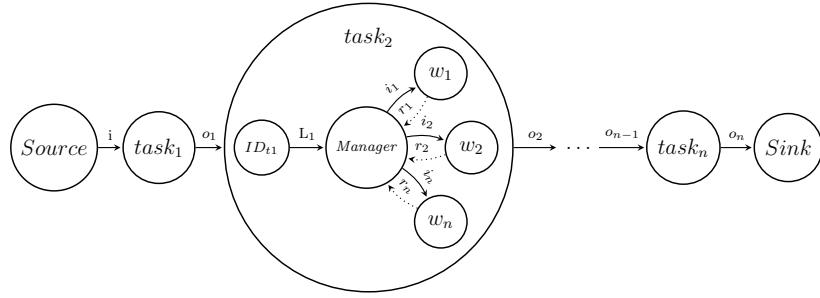


Figure 4: DAG of a workflow including a parallel pattern to execute task 2.

a new VC, to execute a command inside the VC, and to stop or remove the VC.

Figure 3 shows a conceptual representation of how the VCs are managed in DagOnStar. A general flow of how the VCs are managed during the execution of a workflow involves four main phases. In the first phase, the VC containers are launched using Docker. The VCs launched with DagOnStar are configured to share a common volume if they were launched on the same machine. After the containers are deployed, the next phase involves tasks being executed inside the container. Then, the results are sent back to DagOnStar in the third phase. Finally, in the last phase, the garbage collector stops and removes the VCs. Workflows with Docker tasks can be deployed locally, remotely, or both locally and remotely. SSH is used to allow DagOnStar to communicate with the remote VCs.

3.4. Parallel patterns, coupling workflow tasks and task orchestration

Virtual containers are not only used to add a portability property to the tasks of the workflows but also to design and build implicit parallel patterns. The parallel pattern used in the DagOnStar approach, namely the Manager/Worker [71], enables developers to establish parallelism without altering/modifying the code of the tasks considered in a given workflow. Figure 4 depicts an example of such a pattern, in which the four steps necessary to build a parallel pattern are identified: i) To create a script that manages all the tasks of a workflow as a single software instance managing each task as a *worker seed*; ii) To clone w times the *worker seed* to create software instances called *workers*. In the example shown in Figure 4, a parallel pattern includ-

Algorithm 1: Manager/Worker pattern execution

```

Input : Number of workers  $W$ , input data  $\mathbf{L}$ , task data source  $T$ , command to execute
          $command$ 
Output: State of the task  $T$ , path to the results  $\mathbf{R}$ 
 $worker\_template \leftarrow encapsulate(T, command);$ 
 $workers \leftarrow clone(worker\_template);$ 
 $workers.inputs \leftarrow distribute\_load(workers, \mathbf{L});$ 
 $\mathbf{R} = \{\};$ 
 $executors = \{\} state \leftarrow 0;$ 
foreach  $w, i \in (workers, workers.inputs)$  do
     $| executor \leftarrow launchWorker(w, i);$ 
     $| executors[w] = executor ;$ 
end
foreach  $e \in executors$  do
     $| state \leftarrow executor.worker\_state();$ 
     $| \mathbf{R}[w] = executor.getResults();$ 
end
return  $state, \mathbf{R};$ 
  
```

ing workers w_1 , w_2 , and w_3 to execute task 2 is presented; iii) To associate a DagOnStar reserved instance called *Manager* with the workers. This instance is in charge of distributing the workload to these workers and receiving data from either a data source or another task as well as delivering the results to either another task or a data sink. Finally, iv) Encapsulating the instances of a given task into a virtual container (VC). These instances are added to a VC , which represents a single and independent solution. This organization of tasks produces a graph of graphs that is managed by DagOnStar as a single solution (containerized workflow).

Algorithm 1 shows the process to create the Manager/Worker pattern in DagOnStar as well as the coupling of the VCs required to automatically create a workflow in DagOnStar.

In execution time, the parallel pattern manages the workload by executing the following actions: i) the manager reads the input data, creating a list of workload tasks (L) that will be processed by the workers. ii) The manager creates n sub-lists of contents $i_w \in \mathbf{L}$, which include information about the input and output storage scheme assigned by DagOnStar to that task and sends the sub-list of workload tasks to the workers. iii) The workers process the list i_w sent by the manager by reading data from the input storage scheme, write the results in the output storage scheme, and return the processing result status (r_w) to the manager. iv) The manager receives control over the tasks allowing for the delivery of paths of the output storage scheme to the next task (VC) in the workflow.

Algorithm 2: Load balancing.

```

Input : List of workers  $\text{workers}$ , input data  $\mathbf{L}$ 
Output: Workload for each worker:  $\text{workers.inputs}$ 
 $\text{number.workers} \leftarrow |\text{workers}|;$ 
 $\text{workers.inputs} \leftarrow \{\};$ 
foreach  $i \in \mathbf{L}$  do
    do
         $\text{choice}_1 \leftarrow X \sim \mathcal{U}(|\text{number.workers}|);$ 
         $\text{choice}_2 \leftarrow X \sim \mathcal{U}(|\text{number.workers}|);$ 
    while  $\text{choice}_1 \neq \text{choice}_2;$ 
    if  $\text{workers.inputs}[\text{choice}_1].\text{loadsize} > \text{workers.inputs}[\text{choice}_2].\text{loadsize}$  then
         $\text{workers.inputs}[\text{choice}_2] \leftarrow \text{workers.inputs}[\text{choice}_2] + i;$ 
         $\text{workers.inputs}[\text{choice}_2].\text{loadsize} \leftarrow \text{workers.inputs}[\text{choice}_2].\text{loadsize} + i.\text{size};$ 
    else
         $\text{workers.inputs}[\text{choice}_1] \leftarrow \text{workers.inputs}[\text{choice}_1] + i;$ 
         $\text{workers.inputs}[\text{choice}_1].\text{loadsize} \leftarrow \text{workers.inputs}[\text{choice}_1].\text{loadsize} + i.\text{size};$ 
    end
end
return  $\text{workers.inputs};$ 

```

The manager uses a modified version of the *Two Choices random algorithm* [58] (an algorithm traditionally used in storage systems) to distribute the workload to the workers, as presented in Algorithm 2.

3.5. The garbage collector

The role of the DagOnStar garbage collector is to track the storage and computational resources allocated during the execution of tasks and proceed to dispose of them when they are no longer needed.

In case of an *external software component* being executed on the local machine or a remote machine sharing the same file system, when the task scratch directory is no longer needed (and all depending tasks are completed) the directory is removed, making the temporary storage available for new computations.

In the case of *cloud tasks*, the garbage collector takes care of virtual machine stop or termination. If the same virtual machine stance must be used by a single task or multiple ones, advanced disposing policies can be enforced. In a pay per use scenario, where the account is billed on an hourly basis, the policies can even manage billing issues using customized logic provided by the developer. When a virtual machine instance is no longer needed and there are no conditions or enforced policies overriding default rules, the garbage collector stops the virtual machine instance, and eventually terminates it.

For *containerized tasks*, the garbage collector behaves similarly as with

cloud tasks. The difference is related to situations in which the container wrapping the task is contained.

The garbage collector leverages a different management policy when a task is running in *continuous mode*. Because such tasks never end until the workflow ends, or an end condition is verified, each task cycle uses the same scratch directory (this approach is taken to support the common use case where a task runs remotely on a low-power device, where file system operations could be critical due to the kind of local storage).

3.6. DagOnStar Service

As depicted in Figure 1, the DagOnStar Service is a software component not belonging to the DagOnStar runtime (delivered and embedded in the DagOnStar library). This component is devoted to monitoring tasks interacting with the DagOnStar runtime using a REST API. This service can be used to track the status of a workflow or a task of a specified workflow, as described in the Section 4 regarding the DagOnStar application life-cycle.

The service publishes a directory of active workflows and enables the DagOnStar application developer to implement interactions between different workflows of the same application or belonging to diverse applications using the Web tasks. A simple portal enables the users to check the workflow's overall execution status, tasks success and failures, resource allocation, and other insights. The DagOnStar Service has been designed with the idea of managing the user authentication and authorization, the resource allocation, and the “pay as you go” issues.

4. Application Lifecycle

The programming model is defined in such a way that it enables the embedding of one or more workflows in regular Python applications (see Section 4.1). To provide a detailed description of the life-cycle, the interaction of the application with the DagOnStar Runtime was considered (Section 4.2) in addition to the interaction with the DagOnStar Service (Section 4.3).

4.1. Programming model

The DagOnStar programming model enables the developer to create the workflow application as an aggregate of customized task classes extending the ones provided by the framework. The Listing 1 implements a simple parallel data-flow generating 5 random numbers, then a task concatenates

the numbers in one file and finally, another task performs the sum (this example can be compared with a similar circumstance in Parsl).

```

1 import dagon
2
3 class Generate(Batch):
4     def __init__(self, name):
5         Base.__init__(self, name,
6                         command="echo $(( RANDOM )) &> out.txt")
7
8 class Concatenate(Batch):
9     def __init__(self, src[]):
10        Base.__init__(self, "Concatenate",
11                      command="for f in "+" ".join(src)+";do "+
12                          "cat $f >> all.txt; done")
13
14 class Total(Task):
15
16     def __init__(self, src):
17         Base.__init__(self, "Total")
18         self.src=map_input_as_text(src)
19
20     def kernel():
21         total = 0
22         for line in self.src.splitlines():
23             total += int(line)
24         map_result(total, "result.txt")
25
26 ...
27 workflow=Workflow("parallel_dataflow_app",settings)
28 srcURIs=[]
29 for i in range(0,5):
30     srcURIs.append(workflow.add_task(Generate(i)) + "/out.txt")
31
32 concatURI=workflow.add_task(Concatenate(srcURIs))
33 workflow.add_task(Total(concatURI + "/all.txt"))
34 workflow.run()

```

Listing 1: An example of workflow definition using dataflow references and class inheritance from the Batch class.

The task **Generate** is implemented, extending the Batch task and overriding the constructor. Similarly, the task **Concatenate** extends the Bash task adding a list of `workflow://` schema URIs as arguments. Finally, the task **Total** is implemented in the Task class and executed locally. In this case, a mapping between the `workflow://` schema URI of the input file and a local variable is performed. The task algorithm must be implemented to override the `kernel()` method. The result is mapped on a `workflow://` schema URI (`workflow:///total/result.txt`).

Comparing this workflow development approach with the Parsl implicit method, DagOnStar reduces the number of overall lines of code and enforces the Object-Oriented style. Meanwhile, the implementation of branches and

loops in DagOnStar is only possible if dynamically created workflows have already been nested.

4.2. Runtime

The DagOnStar Runtime performs the following basic tasks:

Create Scratch Directory. A scratch directory is created when an external software component must be executed on a local or remote machine. This operation ensures the software component has its isolation at the best of the chosen execution context.

Stage in. This operation is managed by a Stager component when the `workflow://` schema is enforced to define transfer operations in a workflow. For instance, a stage in operation can be a simple local symbolic link, a copy operation, or a data transfer between remote resources. In a containerized execution environment scenario, DagOnStar builds a shared data volume decoupled from the tasks in the workflow. Thus it is available as an external software component from where tasks write and read data.

Run. This is the crucial task operation. Once the task is completed successfully, the related depending tasks can start their execution.

Remove Scratch Directory. This operation is managed by the garbage collector.

Figure 5 uses a simple application to illustrate the DagOnStar life-cycle. We describe in the following the 10 life-cycle events identified in the figure. At time 1, the system is managed by a DagOnStar Service interaction. At time 2, the local scheduler, SLURM, is ready to execute task **a**. A scratch directory is created: the job representing the task is executed on the selected resource and terminated with success at time 3.

At times 4 and 5, the local scheduler is ready to run the jobs related to tasks **b** and **c** and the related scratch directories are created. Tasks **b** and **c** require data produced by task **a**. The `workflow://` schema references are resolved and the Stager component performs the stage in operation in a concurrent way.

Once data are in the tasks scratch directory, each task is executed, ending with success (times 6 and 7). At time 7, tasks **b** and **c** have completed: the data produced by task **a** are no longer needed and that scratch directory is deleted.

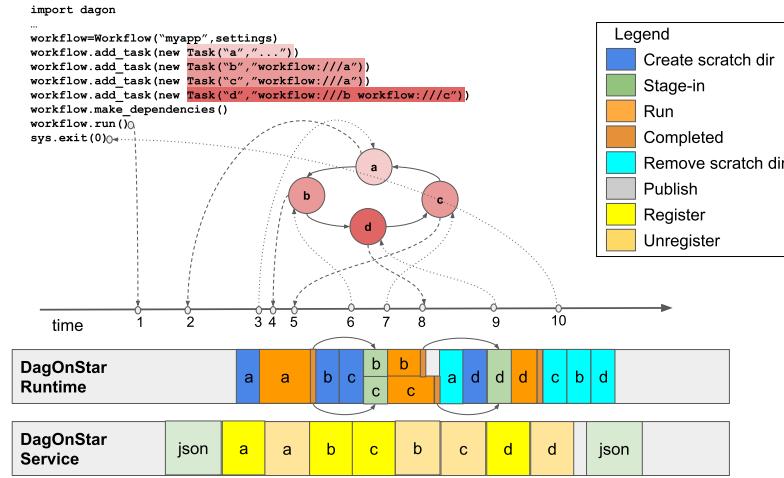


Figure 5: The DagOnStar programming model and the application life-cycle as an interaction sequence between directed acyclic graph parallel job execution, DagOnStar Runtime, and DagOnStar Service.

Meanwhile, the job representing task **d** is ready to be executed by the local scheduler and its scratch directory is created at time 8. Task **d** needs the data produced by tasks **b** and **c**. The references are resolved by using the `workflow://` schema and the Stager component transfers data in the task **d** scratch directory.

When all data have been transferred, task **d** is executed and completed (time 9). Finally, at step 10, the scratch directories of tasks **b**, **c**, and **d** are disposed.

4.3. Service

The interactions between the run-time and the service are performed transparently from the point of view of the application developer using REST APIs. The DagOnStar service performs the following basic tasks:

Publish. The workflow configuration is acquired by the service and made available using REST APIs. This operation is performed both at the workflow execution start and at its termination.

Register. A task registers to the service updating its status each time there is a change in the execution life-cycle.

Unregister. Once a task is completed, it removes itself from the service.

At time 1, the DagOnStar run-time registers the workflow on the DagOnStar service using the JSON representation. This allows the service to provide updates regarding the workflow status. At time 2, task **a** performs its registration to the service and unregisters itself upon completion at time 3. Then, tasks **b** and **c** perform their register/unregister operations between times 4 and 8. Task **d** displays its status changes at times 8 and 9. Finally, at time 10 the DagOnStar run-time interacts with the service, pushing its status as terminated.

5. Use Case: Mexican weather records processing

We implemented a workflow for the acquisition, pre-processing, and classification of meteorological records captured by the Mexican weather station network (EMAS)⁵, covering the 32 states of the Mexican territory [72]. A conceptual representation of this information processing, modeled as a DagOnStar workflow, is depicted in Figure 6.

The *Acquisition* task of this workflow includes a web crawler, which retrieves the weather records from the EMAS website. This crawler downloaded 2 GB of files in a non-structured format in the form of plain text, which was sent forward to the next task (*Preprocessing*).

In the *Preprocessing* task, the retrieved records were processed by a parsing service that transforms these records into files with a structured format. This transformation service executes sub-tasks such as, removing outliers and calculating the missing values. The criteria to remove outliers refers to non-typical values below or above a range of [-40 to 100] °C, whereas a regression based on the values known in the historic registers is used to calculate missing values and to enrich missing data.

In the *Integration* task, the data processed in the previous task were consolidated in a single database, where the mean and median values of each weather metric (minimum temperature, maximum temperature, and

⁵<https://smn.cna.gob.mx/es/observando-el-tiempo/estaciones-meteorologicas-automaticas-ema-s>

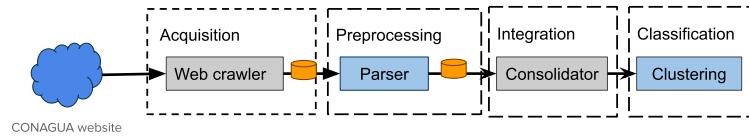


Figure 6: Weather records workflow processing.

precipitation) per EMAS station were calculated for each year captured by these stations.

In the *Classification* task, we implemented a distributed K-Means algorithm [72] to group antennas for different spatial values depending on the similarity of the weather metrics stored in the database created in the previous task.

This workflow is executed as a combination of off-line and online processes. Specifically, the first three tasks are executed in an off-line manner, whereas the classification task is executed on-demand and online. The classification task also includes a processing pipeline with three pre-processing phases. The first one is a search for stations placed at the polygon selected by the end-users. In the second task, the metrics of selected stations are distributed to a cluster of virtual DagOnStar containers to arbitrarily identify metric similarity. In the last task, a list of antennas belonging to the set of groups is created and sent to a geoportal to show the station groups on a map by using a GIS service.

To improve the performance of the tasks of the workflow depicted in Figure 6, *Manager/Worker* parallel patterns were developed to execute the DagOnStar virtual containers. For instance, the service executed in the acquisition task was implemented by using this pattern. In this task, the crawler service was transformed into a pipeline of two systems, the first for finding URLs in the EMAS’ web page that enables users to retrieve the antennas’ data and the second included a retrieval procedure that obtains data from the EMAS web page by using the finding URLs. The two tasks of this pipeline were encapsulated into two virtual containers images. The images also include DagOnStar control software instances such as *Manager* and *Worker*. The *Manager* instance included the URL finding functions, whereas the worker included the retrieval procedures of the web-crawler (See acquisition task in Figure 7). The manager creates as many virtual containers as workers defined in the pattern configuration. When the w workers

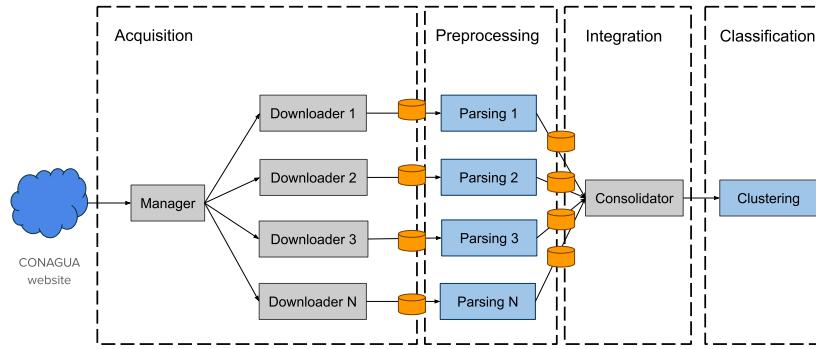


Figure 7: Weather records workflow processing using parallel patterns based on virtual containers.

have been deployed on a given infrastructure, the manager invokes the software instance finding URLs from the EMAS web page, and creates w tasks based on the URL list. The manager distributes the tasks to the workers in a load-balanced way by using a version of the load balancing algorithm presented in [73] but focused on task size instead of file size as established in the original algorithm. The *workers* download the data associated with each task and store these data in a path of a volume shared by all the workers. The chaining of the manager with the workers is performed by using the DAG configuration file created by DagOnStar.

In the pre-processing task, the parser service is also encapsulated into a virtual container image, which is cloned as many times as the workers deployed in the previous task. These workers are chained to the virtual parser container clones in the form of a pipe and filter pattern (see the connection between acquisition and pre-processing tasks in Figure 7). In the processing task, the clustering algorithm was also implemented in the form of a manager/worker pattern. In this task, the manager retrieves all the data available for each station and distributes them to the workers in a balanced manner to calculate the similarity of the climate metrics. This process, therefore, is performed in parallel. The workers deliver the results to a geoportal service that calculates regressions for each cluster identified and for the three available metrics. This service is also executed by using manager/worker patterns (one worker per cluster of stations). This processing scheme enables DagOnStar

to tackle, on-demand and spontaneously, high workloads of requests from end-users and studies of large coverage areas.

The deployment of these patterns by using virtual containers provides the solutions created by DagOnStar with a portability characteristic used to solve issues in real-world scenarios.

6. Experimental evaluation: methodology and results

The case study described in Section 5 is based on information from records captured by the Mexican weather station network (EMAS), which covers the 32 states integrated with the Mexican territory. Each antenna of the EMAS system has registered weather metrics, such as maximum and minimum temperatures (measured in degrees Celsius), as well as the precipitation (measured in millimeters), for 33 years (from 1985 to today). In the *first phase*, the experimental performance was conducted for the evaluation of three different protocols/applications that were added to DagOnStar for the tasks of workflows to exchange data with each other. In this phase, the configuration yielding the best performance when processing the information of the case study was chosen to be used in the next phases. In the *second phase*, parallel patterns were incorporated based on virtual containers to the selected configuration in the previous phase. In this phase, there was also a test of the deployment of DagOnStar on both local and distributed environments by using different configuration settings to identify performance deployment insights. In the *last phase*, the configurations yielding the best performance were tested in both local and distributed scenarios from the previous experiments, and a direct comparison was performed with state-of-the-art workflow engines (Makeflow [40], Parsl [56], and Pegasus [74]).

6.1. Metrics and Test environment

The *response time* was the metric selected to evaluate the performance of tested solutions. This metric represents the time observed by end-users when executing a workflow created by an engine.

The experimental evaluation was conducted by using a test environment composed of two Digital Ocean cloud instances, one Amazon Web Service EC2 cloud instances, and a data cluster on the Cinvestav Tamaulipas in Mexico. Table 3 shows the main characteristics of the virtual machines and containers used in the experimental evaluation.

Compute	Provider	Region	Description	RAM (GB)	Cores	Storage
EC2-1	Amazon EC2	us-west-2	Processing	2 GB	2	Elastic
DigitalOcean-SF	Digital Ocean	San Francisco	Processing	8 GB	8	50 GB
DigitalOcean-London	Digital Ocean	London	Processing	8 GB	8	50 GB
Disys0	Cinvestav Tamaulipas	Tamaulipas, Mexico	Storage/Processing	12 GB	6	1.5 TB

Table 3: Characteristics of the virtual machines and containers.

The description of the solutions evaluated in this case study, as well as details about experiments performed with those solutions, are described in each phase of the experimental evaluation.

6.2. Phase 1: Select data transport solution for distributed environments

We describe experiments designed to identify a suitable policy and solution for the data transport among the tasks of distributed workflows built by using DagOnStar. The basic idea is to show the flexibility of DagOnStar to adapt a data transport mechanism to the I/O systems of the tasks in a workflow. In order to show this property, different data transport applications were incorporated into DagOnStar: secure copy (SCP) [70], Globus [66], and SkyCDS [36]. Four DagOnStar configurations were created: *DagOnStar + SCP*, *DagOnStar + Globus*, *DagOnStar + SkyCDS-Single*, and *DagOnStar + SkyCDS-IDA*.

First, in Table 4, we provide a qualitative comparison of the three data transfer tools (SCP, Globus, SkyCDS) integrated with DagOnStar concerning non-functional requirements such as reliability, security, and efficiency. *Reliability* of data transfer considers whether a solution performs re-transmissions on error events between tasks; reliability of storage considers the ability to withstand failures when data are delivered to the tasks and stored/preserved. *Security* was evaluated in terms of integrity, confidentiality, and access control. Integrity captures whether or not a solution verifies checksums when tasks exchange data in the workflow. Confidentiality was evaluated in terms of preserving the privacy of data end-to-end (data are only available in a clear manner when tasks are processing the data), as the rest of the time the data remains encrypted even when the data are stored. Access control is evaluated in terms of solutions establishing controls (SCP and Globus by using user-password, whereas SkyCDS uses encryption based on attributes). Data movement *efficiency* includes processing as well as cost-efficiency data transport, which is evaluated in terms of solutions applying techniques to reduce the amount of data being transported for large data sets (in the order of gigabytes or terabytes).

Table 4: Qualitative comparison of SCP, Globus, and SkyCDS.

Solutions	Reliability in		Security			Efficiency
	Transport	Storage	Integrity	Confidentiality	Access Control	
DagOnStar + SCP	✓			✓	✓	
DagOnStar + Globus [75]	✓	✓	✓	✓	✓	✓
DagOnStar + SkyCDS [36]	✓	✓	✓	✓	✓	✓

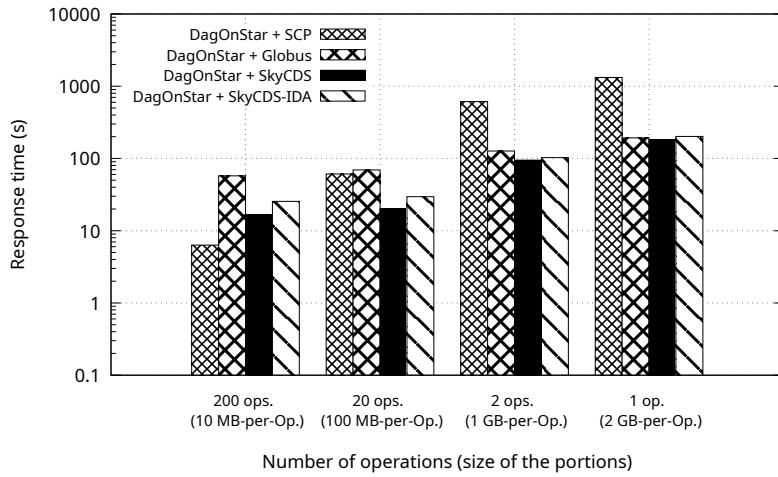


Figure 8: Response time produced by DagOnStar workflows when varying the size of the input data-set and by using different data transfer tools.

Next, we perform a quantitative comparison by running the same experiment multiple times while varying the data transfer protocol. To consider specific policies in this quantitative evaluation, a set of experiments considering different transportation policies was evaluated. In this context, the portion of 2 GB of the EMAS dataset was split into s subsets that were moved from EC2-1 to DO-2. To evaluate the impact of such splitting on transport, we considered not only 1×2000 MB but also 2×1000 MB, 20×100 MB, and 200×10 MB. Intuitively, a trade-off could be expected between the number of operations performed by a transport mechanism and the size of the data units.

The execution times for each of the four configurations *DagOnStar +*

SCP, *DagOnStar + Globus*, *DagOnStar + SkyCDS-Single* and *DagOnStar + SkyCDS-IDA* was measured for $s = 1, 2, 20$, and 200 . Figure 8 shows the results of these experiments. The size of the data unit used in each policy as well as the operations performed by each studied solution resulted in both keys in the resultant performance of the workflows. As it can be seen, the more suitable policy for transporting from task to task in a distributed workflow is the usage of small data units (10 MBs).

The more suitable transportation solution for DagOnStar depends on non-functional requirements to be achieved by a workflow. For instance, in terms of performance, it was observed that for small data units (10 MB and 100 MB) SCP produced better performance than the other configurations. When it is not feasible to choose a policy where a data portion must be transported without splitting it (e.g., 1 GB and 2 GB) or when managing large data units in big data scenarios, that is the case of scenarios studied in this paper, the performance of SCP is degraded and both Globus and SkyCDS-IDA deliver a better fulfillment of efficiency, security, and reliability requirements. As a result of these observations, for big data scenarios, *DagOnStar + Globus* is a trade-off between efficiency, reliability, and improved performance using local connectors and a native APIs [75].

6.3. Phase 2: Evaluate performance of *DagOnStar* processing workflows

In this phase, the performance of processing tasks of a *DagOnStar* workflow is described in terms of deploying the workflow within diverse infrastructures. The workflow described in Section 5 was deployed using four configurations.

Local sequential workflow (LSW). This workflow was launched on a single machine. All the tasks deployed in this workflow are sequentially executed. A physical machine was used to deploy the virtual containers of this workflow (see *Disys0* in Table 3).

Local parallel workflow- N (LPW- N). The tasks of the studied workflow were deployed using parallel patterns built with virtual containers. The workflow depicted in Figure 7 shows different versions of this configuration, which were evaluated by varying the number of workers N in the parallel pattern.

Distributed sequential workflow (DSW). This workflow was deployed in a distributed environment: acquisition on a DigitalOcean-SF in-

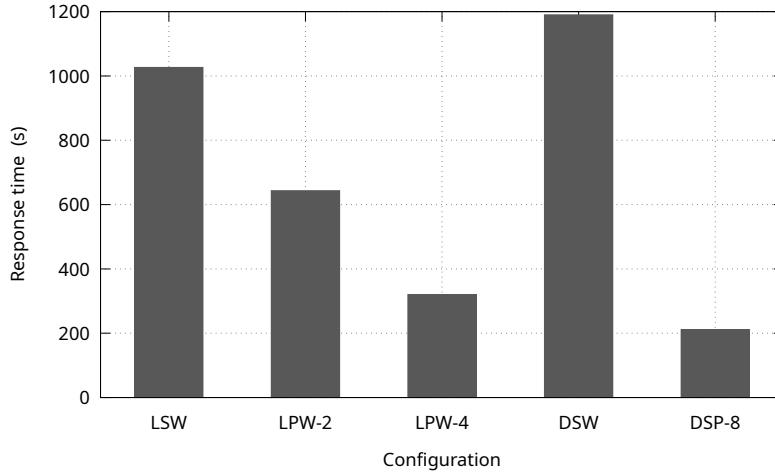


Figure 9: Response time observed for the execution of the workflow deployed by using different configurations of DagOnStar.

stance, preprocessing on a DigitalOcean-London instance, integration in a US West EC2 instance, and clustering on Disys0 (México).

Distributed parallel workflow-N (DSP-N). This configuration combines LSW with DSW into a new configuration. Specifically, the workflow was deployed on a distributed environment, running each task on the same machines of DSW but each task was deployed by using a parallel pattern including as many virtual containers as workers (similar to the LSW configuration).

From Figure 9, we can see that when deploying the tasks using a container-based parallel pattern, the response time of the workflow is reduced. For instance, the response time of LSW was 1026.35 seconds, while with LPW-8, it was 320.36 seconds, indicating an improvement of 68.7% in response time, which was caused by the implicit parallelism produced within the virtual container.

When analyzing the distributed versions (DSP-N) for eight parallel containers for each task in the workflow (DSP-8), an improvement in the response

time of 82.23% was observed in comparison with the execution of the sequential workflow (DSP). This improvement was produced both because of the implicit parallelism inside the virtual container of each distributed worker and the preparation of data performed by DagOnStar before data transportation, which includes an implicit task of data compression for reducing the amount of transported data through the network.

6.4. Phase 3: Compare DagOnStar with state-of-the-art workflow engines

In this section, we compare the performance achieved by DagOnStar with that of three other workflow engines, *Parsl*, *Makeflow*, and *Pegasus*. In each case, we implement the workflows by following the specifications of these different engines in terms of available data transport tools (e.g., FTP, HTTP, Globus, SCP) and data management resources (e.g., Condor [57] and Slurm), with workflows deployed in the physical and virtual machines used in previous experiments (see Section 6.1). Figure 10 depicts the configurations and external tools used for the deployment and execution of the workflows by using the evaluated engines (DagOnStar using Globus for data transfer (LMW-8), Parsl, Makeflow, and Pegasus). To configure the workflow engines, we followed the instructions and information provided by their official documentation (see this guide in usability Figure 12 in Section 6.5) without any form of optimization.

Parsl (MT=8) is a configuration that launches as many threads as cores (8 cores) for executing the tasks of this workflow. Moreover, Slurm was configured to manage the distribution of tasks over the resources available for execution, and the data transfer between machines was performed by using the same Globus version and configuration evaluated in phase 1 (Section 6.2). **For Pegasus**, two configurations were tested: *i*) *Pegasus* as a stand-alone configuration without external tools, and *ii*) *Pegasus + Condor (CC = 8)*, a parallel version implemented by using a Condor pool [57]. Also, SCP was used for the transportation of data through the computational resources in these workflows.

Two Makeflow configurations were tested in these experiments by using SCP as data transfer tool: *i*) *Makeflow (MT=8)*, a version that uses concurrent tasks with eight threads, and *ii*) *Makeflow (MC=8, MT=8)*, a version that uses multi-core parallelism tasks by using eight cores and eight threads. For these two configurations of Makeflow, HTCondor was configured to manage the distribution of tasks over the available resources.

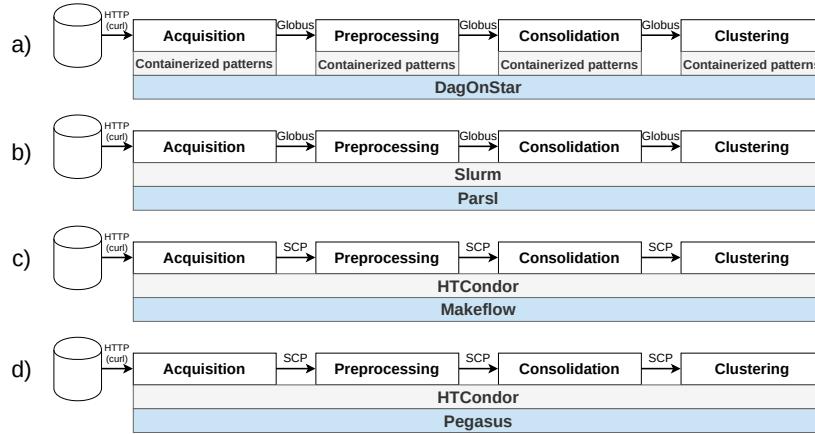


Figure 10: General setup of workflow engine configuration used for experimentation.

Figure 11 shows the response time produced by two configurations of DagOnStar: a standalone version and a manager/worker parallel pattern including eight workers encapsulated into the same number of virtual containers (LMW-8). As demonstrated, the performance yielded by DagOnStar is competitive in comparison with the performance of the other engines. DagOnStar based on patterns (with eight workers, i.e. *DagOnStar LPW-8*) achieved a lower response time by 8.01%, 23.06%, and 2.84% in comparison with Parsl (MT=8), *Makeflow + HTCondor* (MC=8, MT=8), and *Pegasus + HTCondor* (CC = 8), respectively.

When comparing Parsl using 8 threads (MT = 8) with DagOnStarLPW-8, the latter yielded a performance improvement of 18.96%, whereas DagOnStarDSP-8 produced an improvement of 50.19% in the same comparison.

These results only show the behavior of the workflow performance under the configurations and settings chosen from the end-user point of view for the case study conducted in this paper. The results could change when changing any of the configurations, settings, or infrastructure features. For instance, when adding Condor and/or Globus to Pegasus or when changing the connectors for I/O transportation in Globus for Parsl and DagOnStar. A deep performance comparison evaluating the multiple components and configurations of the tested engines is required to determine the efficiency

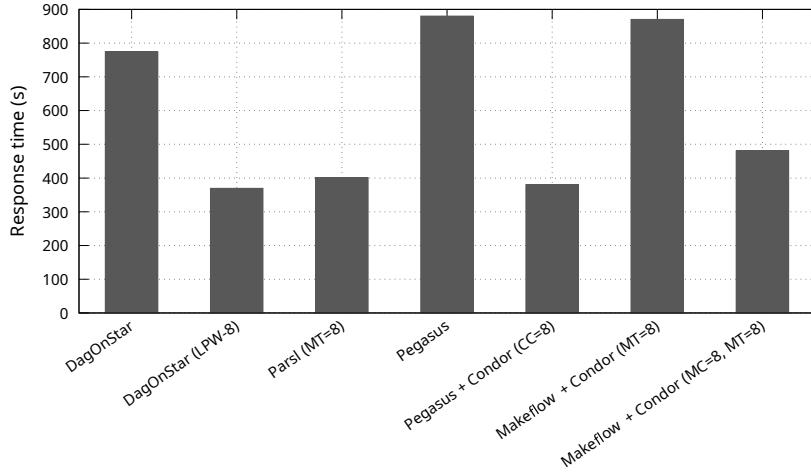


Figure 11: Response time produced by studied workflow engines. MT: Max threads. CC: Condor cores. MC: Makeflow cores.

degree of each engine but this is not the scope of this paper.

6.5. Analyzing the user experience

This section compares the usability of the four workflow engines considered: DagoOnStar, Parsl, Pegasus, and Makeflow. This comparison considers all actions performed by a developer in charge of developing the workflows used in the experimental evaluation of this paper, which was created by using the studied engines.

The actions were organized into pipelines, as it is common practice. Figure 12 depicts in symbols each action required by developers to design, develop, deploy, and execute workflows built by each engine. The large boxes represent actions performed manually, requiring user intervention; the small ones represent actions that are performed automatically by the workflow engine.

DagOnStar is the engine that not only includes more automatic actions but also reduces the number of tasks from the design of workflow solutions to the deployment of that solution within a given infrastructure because this engine does not require the installation of third-party tools.

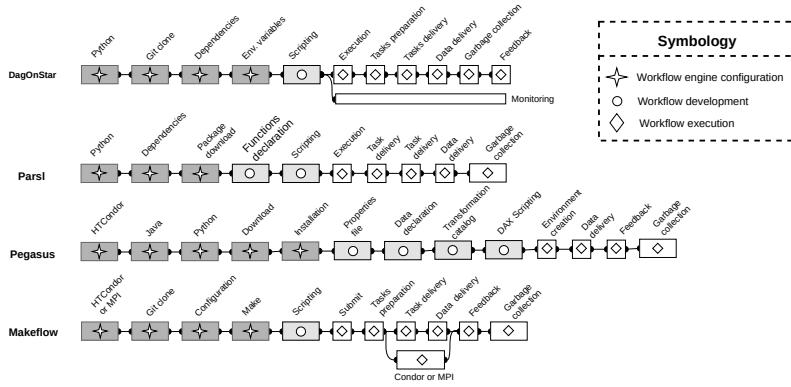


Figure 12: Comparative analysis in the form of pipelines of action required to configure, develop, deploy, and execute a solution by using different workflows engines.

DagOnStar considers control software pieces that transparently and implicitly manage parallelism, load balancing, and task execution as well as the management, delivery, and retrieval of data of the tasks executed at the workflow tasks. In the containerization scheme proposed in DagOnStar, these software pieces are integrated with the tasks executed to create parallel patterns encapsulated into a virtual container. This means that this encapsulation scheme converts a task/workflow into an independent software piece that improves the usability of the solutions as the solutions are ready-to-use software.

This scheme also improves the flexibility of the solutions because the tasks of a workflow are managed as patterns. This ensures that a designer is only expected to deal with a set of task patterns instead of explicit parallel tasks at the workflow level. This means that each task can be configured to manage a different number of threads depending on the resources associated with each container, which improves the management of clones to create more patterns. This feature, finally, also improves the performance of solutions because, in DagOnStar, each task is in charge of managing the threads of all its tasks, which avoids scheduling all the tasks of a workflow (as performed in Parsl and Makeflow). This improves the management of workload queuing.

7. Conclusion and future directions

In this paper, we introduced DagOnStar, a Python-based tool for data-intensive scientific workflows. The DagOnStar programming model enables advanced users to embed workflows in existing Python scientific applications.

We described the DagOnStar architecture design in Section 3 and the programming model and the application life-cycle in Section 4, focusing on its peculiar features such as dependency management via the `workflow://` schema and the garbage collector. We also described the DagOnStar Service, which monitors the execution of the workflows and their tasks. This component records each workflow executed and is in charge of relaunching any task in case of failures. We extended DagOnStar with task parallel patterns based on virtual containers to improving the performance of the tasks of a workflow by replicating n times a virtual container (workers) and distributing the work in a load-balanced manner between the virtual containers. In Section 5, we presented a case study involving the acquisition, pre-processing, and processing of records captured by the Mexican weather station network. Through this experiment, the fine-tuning of DagOnStar data transfer and parallel patterns were described.

This study illustrates the following DagOnStar capabilities:

i) The `workflow://` schema and the integration of different data transportation protocols/applications (here, SCP, Globus, and SkyCDS) for data exchange between tasks in a distributed workflow; *ii)* DagOnStar parallel patterns that improve the performance of workflow replicating tasks over virtual containers, in both local and distributed environments with an improvement in the response time of 82.23% compared with the sequential workflow (DSP); and *iii)* use of virtual containers to launch workflow tasks in a distributing environment, encapsulating each task's source code and its dependencies. Finally, *iv)* we present the results of a direct comparison with Parsl, Makeflow, and Pegasus that reveals that the distributed solution of DagOnStar using a parallel pattern with eight workers yielded an improvement of 50.19 % in response time.

While DagOnStar is being used extensively in different contexts and could be considered a mature product, its development is still ongoing. Short term goals involve adding checkpoint and recovery workflow features, to be implemented in the next development iterations, and improvements in the cloud task interface to enforce the plug-in approach and mitigate effects due to the leveling down of the features offered by diverse cloud providers and

cloud management APIs. From the DagOnStar workflows and containers integration perspective, we have plans to enable running several instances of the external software component in the same container, eliminating the start-up overheads.

In terms of long term development, we are working on the convergence of DagOnStar with GPU applications [76] and with blockchain technologies to establish verifiability and traceability of the transactions performed by different workflow tasks [77].

Acknowledgments

This work is partially included in the framework of the research projects PAUN (ex RIPA PON03PE_00164) and partially supported by the project “Deployable Optics for Remote Sensing Applications – DORA” (ARS01_00653) funded by MIUR – PON “Research & Innovation”/PNR 2015-2020 and by the project “SE4I - AEROMAT - Impiego di tecnologie, materiali e modelli innovativi in ambito aeronautico” (ARS01_01147). founded by MIUR - PON “Research & Innovation”/PNR 2014-2020 (CUP I26C18000140005).

References

References

- [1] D. Di Luccio, G. Benassai, M. de Stefano, R. Montella, Evidences of atmospheric pressure drop and sea level alteration in the ligurian sea, in: 2019 IMEKO TC19 International Workshop on Metrology for the Sea: Learning to Measure Sea Health Parameters, MetroSea 2019, 2020, pp. 22–27.
- [2] S. Torresan, V. Gallina, S. Gualdi, D. Bellafiore, G. Umgiesser, S. Carniel, M. Sclavo, A. Benetazzo, E. Giubilato, A. Critto, Assessment of climate change impacts in the north adriatic coastal area. part i: a multi-model chain for the definition of climate change hazard scenarios, Water 11 (6) (2019) 1157.
- [3] D. Di Luccio, G. Benassai, G. Budillon, L. Mucerino, R. Montella, E. Pugliese Carratelli, Wave run-up prediction and observation in a micro-tidal beach, Natural Hazards and Earth System Sciences 18 (11) (2018) 2841–2857.

- [4] R. Montella, D. Di Luccio, P. Troiano, A. Riccio, A. Brizius, I. Foster, Wacomm: A parallel water quality community model for pollutant transport and dispersion operational predictions, in: Signal-Image Technology & Internet-Based Systems (SITIS), 2016 12th International Conference on, IEEE, 2016, pp. 717–724.
- [5] A. Galletti, R. Montella, L. Marcellino, A. Riccio, D. Di Luccio, A. Brizius, I. T. Foster, Numerical and implementation issues in food quality modeling for human diseases prevention., in: HEALTHINF, 2017, pp. 526–534.
- [6] D. Di Luccio, G. Benassai, L. Mucerino, R. Montella, F. Conversano, G. Pugliano, U. Robustelli, G. Budillon, Characterization of beach run-up patterns in bagnoli bay during abbaco project, *Chemistry and Ecology* 36 (6) (2020) 619–636.
- [7] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, Examining the challenges of scientific workflows, *Computer* 40 (12) (2007) 24–32.
- [8] Y. Zhao, I. Raicu, I. Foster, Scientific workflow systems for 21st century, new bottle or new wine?, in: IEEE Congress on Services, 2008, pp. 467–471.
- [9] D. Stratoulias, V. Tolpekin, R. De By, R. Zurita-Milla, V. Retsios, W. Bijker, M. Hasan, E. Vermote, A workflow for automated satellite image processing: from raw VHRS data to object-based spectral information for smallholder agriculture, *Remote sensing* 9 (10) (2017) 1048.
- [10] J. Goecks, A. Nekrutenko, J. Taylor, Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences, *Genome biology* 11 (8) (2010) R86.
- [11] K. A. Ocaña, D. de Oliveira, E. Ogasawara, A. M. Dávila, A. A. Lima, M. Mattoso, Sciphy: a cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes, in: Brazilian Symposium on Bioinformatics, Springer, 2011, pp. 66–70.

- [12] R. Lovas, G. Dózsa, P. Kacsuk, N. Podhorszki, D. Drótos, Workflow support for complex grid applications: Integrated and portal solutions, in: Grid Computing, Springer, 2004, pp. 129–138.
- [13] I. Taylor, M. Shields, I. Wang, A. Harrison, The Triana workflow environment: Architecture and applications, in: Workflows for e-Science, Springer, 2007, pp. 320–339.
- [14] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al., The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud, *Nucleic acids research* 41 (W1) (2013) W557–W561.
- [15] D. Barseghian, I. Altintas, M. B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. T. Borer, E. W. Seabloom, et al., Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis, *Ecological Informatics* 5 (1) (2010) 42–50.
- [16] S. Gesing, I. Márton, G. Birkenheuer, B. Schuller, R. Grunzke, J. Krüger, S. Breuers, D. Blunk, G. Fels, L. Packschies, et al., Workflow interoperability in a grid portal for molecular simulations, in: Proceedings of the International Workshop on Science Gateways (IWSG10), 2010, pp. 44–48.
- [17] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, et al., Pegasus, a workflow management system for science automation, *Future Generation Computer Systems* 46 (2015) 17–35.
- [18] R. Montella, D. Kelly, W. Xiong, A. Brizius, J. Elliott, R. Madduri, K. Maheshwari, C. Porter, P. Vilter, M. Wilde, et al., FACE-IT: A science gateway for food security research, *Concurrency and Computation: Practice and Experience* 27 (16) (2015) 4423–4436.
- [19] T. J. Skluzacek, K. Chard, I. Foster, Kligmatic: a virtual data lake for harvesting and distribution of geospatial data, in: 2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), IEEE, 2016, pp. 31–36.

- [20] R. M. Badia, J. Ejarque, F. Lordan, D. Lezzi, J. Conejero, J. Á. Cid-Fuentes, Y. Becerra, A. Queralt, Workflow environments for advanced cyberinfrastructure platforms, in: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2019, pp. 1720–1729.
- [21] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. F. da Silva, G. Papadimitriou, M. Livny, The evolution of the Pegasus workflow management software, *Computing in Science & Engineering* 21 (4) (2019) 22–36.
- [22] M. A. Rodriguez, R. Buyya, A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments, *Concurrency and Computation: Practice and Experience* 29 (8) (2017) e4041.
- [23] G. Juve, E. Deelman, Scientific workflows in the cloud, in: Grids, Clouds and Virtualization, Springer, 2011, pp. 71–91.
- [24] G. Laccetti, R. Montella, C. Palmieri, V. Pelliccia, The high performance internet of things: using gvirtus to share high-end gpus with arm based cluster computing nodes, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2013, pp. 734–744.
- [25] J. L. Gonzalez, J. C. Perez, V. Sosa-Sosa, J. F. R. Cardoso, R. Marcellin-Jimenez, An approach for constructing private storage services as a unified fault-tolerant system, *Journal of Systems and Software* 86 (7) (2013) 1907–1922.
- [26] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, *IEEE Cloud Computing* 1 (3) (2014) 81–84.
- [27] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, J. Molina, Controlling data in the cloud: outsourcing computation without outsourcing control, in: CCSW'2009, ACM, 2009, pp. 85–90.
- [28] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, K. J. Eliazar, Why does the cloud stop computing?: Lessons from hundreds of service outages, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, ACM, 2016, pp. 1–16.

- [29] K. Popović, Ž. Hocenski, Cloud computing security issues and challenges, in: The 33rd International Convention MIPRO, IEEE, 2010, pp. 344–349.
- [30] J. Opara-Martins, R. Sahandi, F. Tian, Critical review of vendor lock-in and its impact on adoption of cloud computing, in: International Conference on Information Society (i-Society 2014), IEEE, 2014, pp. 92–97.
- [31] R. Montella, D. Di Luccio, L. Marcellino, A. Galletti, S. Kosta, G. Giunta, I. Foster, Workflow-based automatic processing for internet of floating things crowdsourced data, Future Generation Computer Systems 94 (2019) 103–119.
- [32] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, C. Goble, Why workflows break—understanding and combating decay in taverna workflows, in: 2012 IEEE 8th International Conference on E-Science, IEEE, 2012, pp. 1–9.
- [33] A. Bánáti, P. Kacsuk, M. Kozlovszky, Four level provenance support to achieve portable reproducibility of scientific workflows, in: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2015, pp. 241–244.
- [34] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. DSouza, S. Devoid, D. Murphy-Olson, N. Desai, et al., Skyport-container-based execution environment management for multi-cloud scientific workflows, in: 2014 5th International Workshop on Data-Intensive Computing in the Clouds, IEEE, 2014, pp. 25–32.
- [35] M. Morales-Sandoval, J. L. Gonzalez-Compean, A. Diaz-Perez, V. J. Sosa-Sosa, A pairing-based cryptographic approach for data security in the cloud, International Journal of Information Security 17 (4) (2018) 441–461.
- [36] J. Gonzalez, J. C. Perez, V. J. Sosa-Sosa, L. M. Sanchez, B. Bergua, SkyCDS: A resilient content delivery service based on diversified cloud storage, Simulation Modelling Practice and Theory 54 (2015) 64 – 85.

- [37] D. D. Sánchez-Gallegos, D. Di Luccio, J. L. Gonzalez-Compean, R. Montella, Internet of things orchestration using dagon* workflow engine, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), IEEE, 2019, pp. 95–100.
- [38] M. Cieslik, C. Mura, PaPy: Parallel and distributed data-processing pipelines in Python, arXiv preprint arXiv:1407.4378.
- [39] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, M. Livny, Pegasus: Mapping scientific workflows onto the grid, in: European Across Grids Conference, Springer, 2004, pp. 11–20.
- [40] C. Zheng, D. Thain, Integrating containers into workflows: A case study using MakeFlow, Work Queue, and Docker, in: 8th International Workshop on Virtualization Technologies in Distributed Computing, ACM, 2015, pp. 31–38.
- [41] R. Montella, D. Di Luccio, S. Kosta, Dagon*: Executing direct acyclic graphs as parallel jobs on anything, in: 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), IEEE, 2018, pp. 64–73.
- [42] R. F. da Silva, E. Deelman, R. Filgueira, K. Vahi, M. Rynge, R. Mayani, B. Mayer, Automating environmental computing applications with scientific workflows, in: 2016 IEEE 12th International Conference on e-Science (e-Science), IEEE, 2016, pp. 400–406.
- [43] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D’Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, et al., Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds, in: 2012 4th International Workshop on Modeling in Software Engineering (MISE), IEEE, 2012, pp. 50–56.
- [44] P. Jakovits, S. N. Srirama, I. Kromonov, Viability of the bulk synchronous parallel model for science on cloud, in: International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2013, pp. 41–48.
- [45] C. Liu, D. Zeng, H. Yao, X. Yan, L. Yu, Z. Fu, An efficient iterative graph data processing framework based on bulk synchronous parallel model, Concurrency and Computation: Practice and Experience 32 (3) (2020) e4432.

- [46] A. Barker, J. Van Hemert, Scientific workflow: A survey and research directions, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2007, pp. 746–753.
- [47] L. Chen, Continuous delivery: Huge benefits, but challenges too, IEEE Software 32 (2) (2015) 50–54.
- [48] I. Ascione, G. Giunta, P. Mariani, R. Montella, A. Riccio, A grid computing based virtual laboratory for environmental simulations, in: European Conference on Parallel Processing, Springer, 2006, pp. 1085–1094.
- [49] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, I. T. Foster, Swift/t: Large-scale application composition via distributed-memory dataflow processing, in: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, IEEE, 2013, pp. 95–102.
- [50] M. Abouelhoda, S. A. Issa, M. Ghanem, Tavaxy: Integrating taverna and galaxy workflows with cloud computing support, BMC bioinformatics 13 (1) (2012) 77.
- [51] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock, Kepler: an extensible system for design and execution of scientific workflows, in: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004., IEEE, 2004, pp. 423–424.
- [52] Y. Babuji, A. Brizius, K. Chard, I. Foster, D. Katz, M. Wilde, J. Wozniak, Introducing parsl: A python parallel scripting library (2017).
- [53] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, J. Labarta, Pycompss: Parallel computational workflows in Python, The International Journal of High Performance Computing Applications 31 (1) (2017) 66–82.
- [54] M. Dreher, T. Peterka, Decaf: Decoupled dataflows for in situ high-performance workflows, Tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States) (2017).
- [55] R. Montella, D. Di Luccio, A. Ciaramella, I. Foster, Stormseeker: A machine-learning-based mediterranean storm tracer, in: International

Conference on Internet and Distributed Computing Systems, Springer, 2019, pp. 444–456.

- [56] Y. Babuji, K. Chard, I. Foster, D. S. Katz, M. Wilde, A. Woodard, J. Wozniak, Parsl: Scalable parallel scripting in Python, in: 10th International Workshop on Science Gateways, 2018.
- [57] E. Fajardo, J. Dost, B. Holzman, T. Tannenbaum, J. Letts, A. Tiradani, B. Bockelman, J. Frey, D. Mason, How much higher can htcondor fly?, in: JPCS, Vol. 664, 2015, p. 062014.
- [58] M. Mitzenmacher, The power of two choices in randomized load balancing, *IEEE TPDS* 12 (10) (2001) 1094–1104.
- [59] P. Morales-Ferreira, M. Santiago-Duran, C. Gaytan-Diaz, J. Gonzalez-Compean, V. J. Sosa-Sosa, I. Lopez-Arevalo, A data distribution service for cloud and containerized storage based on information dispersal, in: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), IEEE, 2018, pp. 86–95.
- [60] R. Qasha, J. Caña, P. Watson, A framework for scientific workflow reproducibility in the cloud, in: 2016 IEEE 12th International Conference on e-Science (e-Science), IEEE, 2016, pp. 81–90.
- [61] D. D. Sánchez-Gallegos, D. Di Luccio, J. Gonzalez-Compean, R. Montella, A microservice-based building block approach for scientific workflow engines: Processing large data volumes with DagOnStar, in: 2019 15th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), IEEE, 2019, pp. 368–375.
- [62] E. Afgan, A. Lonie, J. Taylor, K. Skala, N. Goonasekera, Architectural models for deploying and running virtual laboratories in the cloud, in: Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on, IEEE, 2016, pp. 282–286.
- [63] C. Sloggett, N. Goonasekera, E. Afgan, Bioblend: automating pipeline analyses within galaxy and cloudbuild, *Bioinformatics* 29 (13) (2013) 1685–1686.

- [64] R. K. Madduri, D. Sulakhe, L. Lacinski, B. Liu, A. Rodriguez, K. Chard, U. J. Dave, I. T. Foster, Experiences building Globus Genomics: A next-generation sequencing analysis service using Galaxy, Globus, and Amazon Web Services, *Concurrency and Computation: Practice and Experience* 26 (13) (2014) 2266–2279.
- [65] R. Montella, A. Brizius, D. Di Luccio, C. Porter, J. Elliot, R. Madduri, D. Kelly, A. Riccio, I. Foster, Using the FACE-IT portal and workflow engine for operational food quality prediction and assessment: An application to mussel farms monitoring in the bay of napoli, italy, *Future Generation Computer Systems* 110 (2020) 453–467.
- [66] K. Chard, S. Tuecke, I. Foster, Globus: Recent enhancements and future plans, in: *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, 2016, pp. 1–8.
- [67] H. R. Arkian, A. Diyanat, A. Pourkhalili, MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications, *Journal of Network and Computer Applications* 82 (2017) 152–165.
- [68] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, S. Tata, A precise meta-model for open cloud computing interface, in: *8th IEEE International Conference on Cloud Computing (CLOUD 2015)*, 2015, pp. 852–859.
- [69] C. Boettiger, An introduction to Docker for reproducible research, *ACM SIGOPS Operating Systems Review* 49 (1) (2015) 71–79.
- [70] J. Pechanec, How the SCP protocol works, *Weblog Post. Jan Pechanec's Weblog. Oracle.*
- [71] J. L. Ortega-Arjona, *Patterns for Parallel Software Design*, 1st Edition, Wiley Publishing, 2010.
- [72] D. D. Sánchez-Gallegos, J. L. Gonzalez-Compean, S. Alvarado-Barrientos, V. J. Sosa-Sosa, J. Tuxpan-Vargas, J. Carretero, A containerized service for clustering and categorization of weather records in the cloud, in: *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, 2018, pp. 26–31.

- [73] P. Morales-Ferreira, M. Santiago-Duran, C. Gaytan-Diaz, J. L. Gonzalez-Compean, V. J. Sosa-Sosa, I. Lopez-Arevalo, A data distribution service for cloud and containerized storage based on information dispersal, in: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018, pp. 86–95.
- [74] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus: a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35.
- [75] Z. Liu, R. Kettimuthu, J. Chung, R. Anathakrishnan, M. Link, I. Foster, Design and evaluation of a simple data interface for efficient data transfer across diverse storage, <https://arxiv.org/abs/2009.03190>.
- [76] L. Marcellino, R. Montella, S. Kosta, A. Galletti, D. Di Luccio, V. Santopietro, M. Ruggieri, M. Lapegna, L. D’Amore, G. Laccetti, Using gpgpu accelerated interpolation algorithms for marine bathymetry processing with on-premises and cloud based computational resources, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2017, pp. 14–24.
- [77] G. A. Vazquez-Martinez, J. Gonzalez-Compean, V. J. Sosa-Sosa, M. Morales-Sandoval, J. C. Perez, Cloudchain: A novel distribution model for digital products based on supply chain principles, International Journal of Information Management 39 (2018) 90–103.

PuzzleMesh: A puzzle model to build mesh of agnostic services for edge-fog-cloud

Dante D. Sanchez-Gallegos, J. L. Gonzalez-Compean, Jesus Carretero, Heidy M. Marin-Castro, Andrei Tchernykh, and Raffaele Montella

Abstract—This paper presents the design, development, and evaluation of PuzzleMesh, an agnostic service mesh composition model to process large volumes of data in edge-fog-cloud environments. This model is based on a puzzle metaphor where pieces, puzzles, and metapuzzles represent self-contained autonomous and reusable software artifacts encapsulated into containers and published as microservices. A *piece* represents the integration of apps with I/O interfaces (*loops/sockets*), parallel processing, and management software. A *puzzle* represents a processing structure (e.g., workflows) built coupling pieces through loops and sockets. Puzzles integrate structures with a microservice architecture, implicit continuous dataflows, and transparent data exchange management software. A *metapuzzle* represents a recursive assemble of puzzles. A mesh represents a pool of pieces, puzzles, and metapuzzles available for designers to choose artifacts to build services. A prototype developed using PuzzleMesh model was evaluated through case studies about the automatic construction of processing services for the acquisition, pre-processing, manufacturing, preserving, and visualizing of satellite imagery. A qualitative comparison revealed that PuzzleMesh provides a flexible way to build reusable and portable services and to improve the usability of the services. The case study also revealed that PuzzleMesh yielded better performance results than other state-of-the-art tools.

Index Terms—Agnostic services, Big Data Pipelines, Microservices Architecture, Edge-Fog-Cloud Services

1 INTRODUCTION

CLOUD and web services are keystones for organizations to manage the lifecycle of their digital assets (e.g., any of images, documents, contracts, and health data) [1] and digital products (e.g., satellite/healthcare imagery) [2]. A digital product lifecycle traditionally considers stages such as acquisition, preservation, processing, sharing, and visualization [1], [3]. Organizations commonly manage these stages as services [4] that can be coupled in the form of processing structures (e.g., workflows [5], [6], pipelines [7], and patterns [8]), which are created with different frameworks [1], [9]. However, these frameworks establish requirements for using specific infrastructures and platforms that commonly require the installation of third-party solutions to manage the resources where they are deployed or improve their performance (e.g., Slurm or HTCondor [10], [11]). Moreover, the processing structures commonly exhibit a dependency on the infrastructure, especially for data storage and processing. Those dependencies represent an obstacle for organizations to deploy these structures on different types of platforms and infrastructures [1], [4], [12].

- Sanchez-Gallegos and Gonzalez-Compean (corresponding author) are with the Cinvestav Tamaulipas, Cd. Victoria, Mexico, E-mail: {dante.sanchez, jaseluis.gonzalez}@cinvestav.mx.
- Carretero is with the Universidad Carlos III de Madrid, Madrid, Spain, E-mail: jcarrete@inf.uc3m.es .
- Marin-Castro is with Cátedras CONACYT - Universidad Autónoma de Tamaulipas, Cd. Victoria, Mexico, E-mail: lmarin@conacyt.mx.
- Tchernykh is with the CICESE Research Center, Ensenada, Mexico, Ivannikov Institute for System Programming, RAS Moscow, Russia, and South Ural State University, Chelyabinsk, Russia, E-mail: chernykh@cicese.mx.
- Montella is with the Department of Science and Technologies - University of Naples "Parthenope", Naples, Italy, E-mail: raffaele.montella@uniparthenope.it.

As it can be seen, the main problem identified in the composition of services for supporting the lifecycle of digital products is the dependency of services with any of infrastructure [8], platform [13], programming language [14], or third-party services [15]. For instance, engines such as Makeflow and Pegasus requires third party services (e.g., Slurm [16], HTCondor [10] or Amazon WS) to manage the data exchange and the task execution in either HPC or cloud infrastructures. This dependency is produced by a static formalization based on static directed acyclic graphs (DAG) [17] used to compose the structures of services. In DAGs, nodes and edges are defined in design time, whereas, in deployment time, the parameters of a DAG such as the number of parallel workers, sources, and workspace can be fixed either for a given infrastructure or connected to a third-party service [18]–[20]. This DAG formalization creates a dependency between services with any of an infrastructure, platform, or third-party service. A static DAG thus may not be suitable in scenarios such as data science [21], big data [3], [22], and stream processing [23], [24] where the data are processed by multiple applications through multiple stages and infrastructures (e.g., on edge-fog-cloud).

Instead of using traditional static DAG management, in this paper we propose a high-level service composition based on configurable DAGs that are built by the following principles: *i*) a self-contained processing model that encapsulates DAGs and management structures (e.g., load balancers, dispatchers, data delivery/retrieval) into autonomous and independent software pieces. This avoids dependencies of services with a given infrastructure/platform. *ii*) Implicit parallelism as well as data and task distribution, which enables designers to create scale in/out strategies in development time. This avoids dependency of services

with the programming models and third party services, as the pieces are in charge of data management and scale operations. *iii)* a recursive coupling of pieces enables designers to connect a DAG of a given piece with DAGs of other pieces. This enables designers to reuse existent services, removing/adding pieces from/to a service, or to create complex solutions without stopping the involved services. This avoids the dependency of services with the applications included in these services.

The union of these principles into a single service composition puzzle-based model provides the services with *agnosticism* property. This property enables organizations, at a high level, to design, implement and manage services on heterogeneous infrastructures without suffering from side effects of vendor lock-in [25]. This property enables organizations, at a high level, to design, implement and manage services on heterogeneous infrastructures without suffering from side effects of vendor lock-in [25]. Agnosticism is a complex property requiring the meeting of non-functional requirements (*NFR*) such as portability, re-usability/usability, efficiency, security, and reliability to name a few. With this property, organizations have the control of the service over its development and execution, as well as data exchange and even management required to create continuous dataflows connecting users with the services [24].

Three challenges arise when trying to add agnosticism to the services: *i)* to create a service composition that, in design time at a high level, enables designers to add as many NFR as required to ensure the building of continuous dataflows; *ii)* to materialize, in development time, the design of a service that automatically manages dataflows on heterogeneous infrastructures; and *iii)* to enable, in execution time, implicit management of data exchange and events for services to avoid dependencies with any of programming-language, platform, and infrastructure.

Portability, heterogeneity, reusability/usability, and efficiency are examples of properties that should be considered when creating agnostic solutions. *Portability* is required for deploying services on multiple infrastructures [26] [25]. *Heterogeneity* management is required for building services by using multiple apps developed with different programming languages and frameworks [5], [27]. Reusability/usability of services is required for taking advantage of already tested and installed software [28] as well as for end-users and/or other services can online consume/use these services in a straightforward manner. Efficiency is required for services to manage and process large volumes of data [7], [22], [29] for critical decision-making procedures.

In this paper, we propose PuzzleMesh, a service mesh model based on a puzzle metaphor that enables designers, organizations, and/or end-users to build services based on comprehensive processing structures to manage large volumes of digital information assets/products throughout their lifecycle, which can be deployed on different scenarios considered in edge-fog-cloud environments.

This model includes two basic components. The first one is a *self-containing method* that adds the *agnosticism* property to the services built by using PuzzleMesh. This property represents that our solution provides services with properties such as portability, heterogeneity management,

reusability/usability, and efficiency. The second component of this model is a *recursive service composition based on a puzzle metaphor* that enables designers, organizations, or end-users to create processing structures to manage large volumes of data by interconnecting sets of software artifacts.

The integration of these components is achieved in the form of artifacts called pieces, puzzles, and metapuzzles. A *piece* represents the integration of apps with I/O interfaces (*loops/sockets*), parallel processing, and auto-management software. These pieces can be interconnected with other pieces by using embedded *sockets* and *loops* (I/O interfaces of the pieces respectively). A *puzzle* represents a processing structure (e.g., a pipeline) built by coupling pieces through loops and sockets managed as a *DAG*. Puzzles also integrate these structures with a microservice architecture, implicit continuous dataflows, and transparent data management software. A *metapuzzle* represents a recursive assembly of combinations of puzzles.

All piece artifacts (pieces, puzzles, metapuzzles, or any combination of them) are incorporated into a *mesh* controlled by PuzzleMesh. This means that for designers to build processing structures and services they only choose piece artifacts from the mesh, define their coupling, assign them data sources/sinks (e.g., cloud storage locations or filesystems), and assign them infrastructure resources (e.g., virtual/physical machines). This is feasible because, all the artifacts include tools for the orchestration, choreography, launching, and coupling included in the artifacts.

Our proposal has several advantages. On the one hand, the self-contained method ensures an automatic deployment of the services through different infrastructures, reducing troubleshooting tasks and mitigates the risks of suffering from vendor lock-in. On the other hand, the puzzle metaphor produces a flexible composition of services, which is quite useful for organizations to create processing structures for processing large volumes of data.

To evaluate the PuzzleMesh model, we conducted a case study based on the management of satellite imagery captured by the ERIS ground station [30], as well as the manufacturing, and visualization of the products in a geoportal service. The experimental evaluation performed in the case study revealed that the proposed model yields better performance results than other state-of-the-art solutions.

The main contributions of this work are the following:

- A self-containing method for building infrastructure-agnostic processing structures. This model not only enables designers to reuse processing structures in a lifecycle and adds portability to the applications, but it also represents a register of the decisions taken at design, development, and execution time to provide services with agnosticism property, which results key to compose services in edge-fog-cloud environments.
- A puzzle metaphor model for the construction and management of independent/autonomous, reusable, portable, and efficient services to process large volumes of data during the lifecycle of digital products.
- An automatic and transparent dataflow control based on an implicit parallelism scheme and load-balancing for the continuous data processing and delivery.

TABLE 1: Qualitative evaluation of PuzzleMesh, frameworks and engines to build processing structures.

Work	Scope			Agnosticism level (1) (2) (3)	Coupling and execution management	Non-functional requirements			
	Design-oriented	Deployment-oriented	Agnostic			Eff.	Reu.	Por.	Re.
Taverna [31]	-	-	-	-	Centralized	-	*	-	-
Comps [32]	-	-	-	-	Centralized	-	*	-	-
Jenkins [33]	-	-	-	-	Centralized	-	*	-	-
GlobusGalaxies [34]	-	-	-	-	Centralized	-	*	-	-
Parsl [18]	-	-	-	-	Centralized	-	*	-	-
Makeflow [35]	-	-	-	-	Centralized	-	*	-	-
Pegasus [20]	-	-	-	-	Centralized	-	*	-	-
DagOnStar [19]	-	-	-	-	Centralized	-	*	-	-
Istio [36]	-	-	-	-	Self-contained	-	*	-	-
OpenShift [37]	-	-	-	-	Centralized	-	+	-	-
Slurm [16]	-	-	-	-	Centralized	-	*	-	-
HTCondor [10]	-	-	-	-	Centralized	-	*	-	-
PuzzleMesh	-	-	-	-	Self-contained	-	*	-	-

* refers to reusability of specific configurations and deployment files.

+ refers to re-usability of the apps and/or stages previously created by the user or other users.

(1) Language-agnostic; (2) Platform-agnostic; (3) Infrastructure-agnostic.

Eff.: Efficiency; Reu.: Reusability; Por.: Portability; Re.: Reliability; Inter.: Interoperability

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes a model based on a puzzle metaphor. Section 4 presents the implementation of a prototype based on the model. Section 5 describes the experimental results from a case study. Conclusions and future research lines are described in Section 6.

2 RELATED WORK

Processing structures such as workflows, design patterns and pipelines are commonly used to support the management of the digital products life cycle in the edge, the fog, or the cloud [23], [24]. Table 1 shows a summary of different frameworks, classified according to their scope to create structures (oriented to the design or deployment), the degree of agnosticism (programming language, platform, or infrastructure) and the NFRs managed (efficiency, portability, reliability, and reusability) by the studied solutions.

Design-oriented solutions are primarily focused on the design of services based on the coupling applications at high level. These solutions provide end-users with either programming-models or declarative schemes [17]. For example, Parsl and DagOnStar provide end-users with a Python programming model to design workflows based on DAGs. Makeflow and Pegasus also provide end-users with interfaces to write DAGs' configuration files. These solutions rely on third-party tools to perform the management of the resources and/or the tasks scheduling, which creates a dependency with these tools. For example, Makeflow and DagOnStar delegates the scheduling of tasks to HTCondor [10]/Slurm [16] when working with HPC clusters. In turn, deployment-oriented solutions such as schedulers (e.g., HTCondor), workload managers (e.g., Slurm), and cloud APIs (e.g., AWS) perform the scheduling and distribution of processing structures through environments such as cloud and HPC clusters. These solutions commonly are developed for a given infrastructure or platform (e.g., Slurm and HTCondor are only available for Linux clusters). Nevertheless, this could generate a dependency problem, making costly and difficult the transition of structures, data, and applications from one platform/infrastructure to another one [15].

Instead of traditional solutions, PuzzleMesh considers an integration designing-oriented and deployment-oriented solutions into a single solution. In this way, PuzzleMesh keeps the design at high level (by using puzzle piece metaphor) and an automatic deployment and management (by using the self-contained and recursive coupling models), which provides services with agnosticism property. When

Management level	Hierarchy elements	Causes-issues	Solution
Life cycle	Stages	Management of different life cycles	Service mesh model.
Stages	Stage 1 Stage 2 Stage 3	Heterogeneity and interconnection	Puzzle abstractions.
Applications	App 1 App 2 App 3	Efficiency issues due to apps heterogeneity	Parallel patterns and load balancing schemes.
Environment	Inputs Dependencies Outputs	Portability and reusability problems.	Encapsulation scheme.

Fig. 1: Hierarchical levels managed during the digital products lifecycle.

providing services with agnosticism, the dependency of these services with any of the language [14], infrastructure [8], or platform [13] can be mitigated or eliminated. In this section, we perform a qualitative study comparing the agnosticism level produced by solutions available in the state-of-the-art.

Language-agnostic solutions (e.g., Pegasus, and Makeflow) interconnects heterogeneous applications developed by using different programming languages and tools [14]. These solutions commonly depend on any of a specific platform, infrastructure, or third-party tool. Platform-agnostic solutions, (e.g., DagOnStar and Istio [36]) creates portable services, which however are commonly deployed on a single environment (any of the edge, the fog, or the cloud). In turn, infrastructure-agnostic solutions (e.g., Comps [32]) enable the interconnection of multiple environments [8]. Instead of traditional solutions, PuzzleMesh model provides services with language, platform, and infrastructure agnosticism in an integral. To achieve this type of agnosticism, it is required meeting NFRs such as efficiency, reliability, portability, and reusability.

Parsl, DagOnStar, and Jenkins implement thread-based parallelism to efficiently process data, whereas Pegasus or Makeflow invoke external tools (e.g., MPI). In turn, PuzzleMesh combines patterns of threads with load-balancers to concurrently and implicitly execute apps without installing third-party solutions because of its self-contained model. PuzzleMesh, Makeflow, Pegasus, and Comps provides services with portability by using virtual containers. Nevertheless, the services built with PuzzleMesh self-contained model. This means the pieces do not depend on a central service, as is the case of the other solutions.

Reliability is crucial to make accessible the data and applications even during outages. Istio [36] and OpenShift [37] can recover applications from the last state before a failure, but not from data unavailability. PuzzleMesh prepares data by using Information Dispersal Algorithm (IDA) [38] to recover unavailable data.

3 PUZZLEmesh: A MESH MODEL BASED ON A PUZZLE METAPHOR

In this section, we define the problem-solving process for PuzzleMesh to create structures for processing large volumes of data deployed on edge-fog-cloud scenarios. The second part of this section presents the design principles of PuzzleMesh defined for solving the identified issues.

3.1 Establishing problem-solving: causes, issues, and solution

Figure 1 depicts issues that arise when building structures for processing large volumes of data in edge-fog-cloud environments during the digital product lifecycle. We define a

hierarchical representation with 4 levels: the digital product lifecycle, lifecycle stages, apps deployed on each stage, and the environment where the apps are deployed.

At the first level, organizations and designers should manage different lifecycles of their digital products [1], [4]. This includes the definition of functional (FR) and non-functional (NFR) requirements (e.g., adding security levels in the case of sensitive data [39], or traceability for managing contracts [40]). At this level, PuzzleMesh proposes establishing a service mesh from where designers can create comprehensive solutions for different types of digital product lifecycles. Designers can add, remove, and update stages of a lifecycle in a flexible manner [5], [27].

At the second level, designers find *heterogeneity* and *interconnection* issues of the apps required for each stage. Apps are developed for different platforms and should be interconnected with other apps at other stages [5], [27]. Continuous delivery and continuous integration (*CD/CI*) [33] paradigms face up this issue to avoid downtime or delays when performing coupling and integration tasks. Nevertheless, it is not always feasible that all apps establish the data exchange through the stages. PuzzleMesh proposes self-contained structures (pieces, puzzles, and metapuzzles) that produce agnostic services in charge of the interconnections with other structures by using implicit management microservices. Moreover, these self-contained structures decouple the I/O calls from the apps and relegate this task to the pieces, which means that the designers can re-configure those connections even in execution time.

At the third level, designers could find efficiency issues produced by the heterogeneity in the performance of the apps [7] due to bottlenecks typically produced in workflows used in big data [22], [41]. In such scenario, additional parallelism managers (e.g., HTCondor [10] or MPI [42]) are commonly incorporated to frameworks for improving the management of resources [20], [35]. Nevertheless, this type of addendum could impose a given platform or infrastructure as an additional requirement, which could return the designers to the issues arising at the stage level [25]. Instead, PuzzleMesh incorporates, implicit management of parallel patterns for each stage to manage its resources.

At the fourth level, portability and reusability problems arise [26], [28] when the input and output data management, as well as the app' dependencies, could be highly tied with the environment where the apps are deployed (e.g., in a vendor lock-in scenario [25]). Containers and microservice architectures are increasingly used by organizations to face up those issues [26]. PuzzleMesh provides portability and reusability of the solutions by using self-contained structures that convert apps into software artifacts encapsulated into containers that are published as microservices.

3.2 Puzzle metaphor description

To face the issues previously identified and to implement the solutions proposed for each level described in Section 3.1, we propose in this paper to use a puzzle metaphor for building infrastructure-agnostic services. In this metaphor, a *piece* of a puzzle represents a software artifact that includes modular components such as an app, management metadata, auto-management tools, sockets/loops, and dependencies. The app represents the functionality of the piece.

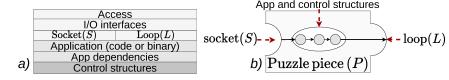


Fig. 2: Architectural stack of: (a) a piece and (b) its conceptual representation.

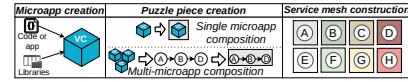


Fig. 3: Conceptual representation of the construction of a service mesh based on the puzzle metaphor.

The management metadata describes information such as the input/output parameters, paths to get data and execute code/binary, and the parameters of auto-management that are required for the app to be executed. The auto-management tools establish controls over the execution of parallel patterns, the load-balancing, and the data management that are performed within the piece. *Sockets* (input interfaces) and *loops* (output interfaces) couple the pieces with other pieces. Those interfaces intercept the I/O requests sent/produced to/by apps and make an indirection to the artifact context, which decouples the apps from storage and CDNs. The *dependencies* include the operating system, libraries, and environment variables required for the app to be executed in the same way in which it was tested and approved to be in production.

To convert an app into autonomous, portable, and reusable artifact software, all the above-described components are encapsulated into a container, which is managed as a self-contained structure. At this point, a *piece* is ready to be deployed on an infrastructure. Figure 2 depicts an architecture stack (a) of a piece (*P*) and its components. The access layer verifies the tokens and credentials for ensuring valid users/apps/pieces accessing the incoming/departing data to/from a piece. A piece (*P*) is interconnected to other pieces by using its sockets and loops (I/O interfaces that can be any of the network, filesystem, or memory). The sockets (*S*) read contents from a data source or another *P*, whereas loops (*L*) write contents to another *P* or a data sink. Loops and sockets implement network interfaces (REST and TCP sockets) because it is expected that these *P*s will be deployed in distributed environments. This model enables pieces to decouple the apps from the management of the data. Figure 2 also shows the conceptual representation of this piece (b). The stack is used in the piece to convert all its components into a black box.

Figure 3 shows the process of encapsulating apps into containers, the composition of multiple apps into pieces, and the creation of a service mesh by storing and indexing the pieces built by the designers and end-users.

In this model, a puzzle represents a processing structure that is built by coupling a set of pieces ($Puzzle = \{P_1, P_2, \dots, P_n\}$), which are chosen from the service mesh. These pieces are chained through the loops and/or sockets by using a directed acyclic graph (DAG), which determines the direction in which data is processed by the pieces. The beginning of a DAG is a data source that is used by the

IEEE TRANSACTIONS ON SERVICES COMPUTING

5

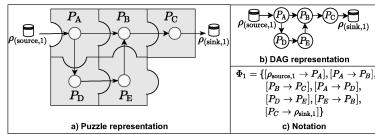


Fig. 4: Conceptual representation of a) a puzzle, b) its DAG representation, and c) its notation.

puzzle to inject data to the first piece in the pattern, whereas a data sink represents the ending of a DAG.

A puzzle artifact is thus a pattern (e.g., pipeline or workflow) suitable for managing a digital product lifecycle. To produce a processing structure as a service, this artifact includes implicit management software implementing a microservice architecture, metadata (DAG), an API Rest used to exchange messages between the P_s of a puzzle, as well as a content delivery network library (CDN) called SkyCDS [43], which delivers digital products to the P_s by using valid catalogs created to give consistency and synchronization to the data exchanges among the pieces considered by a DAG. Figure 4 shows an example of a puzzle. This puzzle includes 5 pieces (P_A, P_B, P_C, P_D , and P_E) connected to a data source ($\rho_{(source,1)}$) that process incoming contents and delivering the results to a data sink ($\rho_{(sink,1)}$). As a result, this puzzle is managed as a service. A metapuzzle represents a pool of puzzles, which is managed as a service mesh, including a puzzle manager for coordinating the puzzles.

As already said, all the PuzzleMesh components are self-contained, which means that pieces, puzzles, and metapuzzles manage the deployment of their components and coupling with other pieces by using the embedded management software added at each construction level.

3.3 A puzzle model piece to create infrastructure-agnostic services

To model a piece (P), we first define the app ($A \in P$) to process data. Thus, a P includes a set of loops (L) and a set of sockets (S), as well as a containerized app, as follows:

$$P = [S, A, L]. \quad (1)$$

where A is a containerized app; S and L represent the input (sockets) and output (loops) interfaces; and $S \wedge L = \{\text{network} \vee \text{memory} \vee \text{file_system}\}$.

We denote the interconnection of 2 pieces in a puzzle as $P_i \rightarrow P_{i+1}$, where \rightarrow is the interconnection of the loop of the piece P_i with the socket of the piece P_{i+1} . Thus, a puzzle (Φ) in the form of a DAG is represented as the following:

$$\Phi = [\rho_{(source,y)} \rightarrow P_1], [P_1 \rightarrow P_2], [P_2 \rightarrow P_3], \dots, [P_{n-1} \rightarrow P_n], [P_n \rightarrow \rho_{(sink,z)}], \quad (2)$$

where n is the total number of pieces in the puzzle (Φ), $\rho_{(source,y)}$ is the source pool from where P_1 acquire digital products, $\rho_{(sink,z)}$ is the sink pool where the contents are written, y and z are identifiers or routes to access to the storage space of the sources and sinks. Figure 4 depicts a graphic representation of a puzzle of 5 pieces, its representation as a DAG, and its notation.

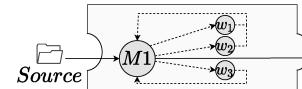


Fig. 5: Conceptual representation of a parallel pattern (manager/worker) embedded in a piece.

3.3.1 Implicit parallelism and load-balancing within pieces

In PuzzleMesh, parallel patterns (e.g., manager/worker and divide&conquer) are created by cloning and chaining *apps* within the *pieces*. These patterns concurrently execute the clones for improving the piece efficiency, which is key when processing large volumes of data. A pattern is denoted as the following:

$$\varphi(P) = M(n, \text{pattern}) \Leftrightarrow \{w_1 \parallel w_2 \parallel w_3 \parallel \dots \parallel w_n\} \quad (3)$$

where $\varphi(P)$ indicates that the piece (P) includes a parallel pattern with n workers, and *pattern* indicates the parallelism pattern to be used (*mw* for manager/worker and *dq* for divide&conquer). M is the manager of the clones; and w_1, \dots, n are clones of the original apps $A_1 \in P_1$. A double arrow (\Leftrightarrow) denotes the distribution of digital products from the manager to the workers.

Figure 5 shows a parallel pattern embedded in a piece created by the previous notation. As can be seen, the manager distributes the load to the workers, which is performed by using a load-balancing algorithm based on two pseudorandom choices [38]. The workers process the assigned data, and the results are delivered to the manager. The manager is in charge of delivering the results to either the next piece in the pattern or to a sink.

Input: List of workers (W) and data to process (d)
Output: Worker (w) to complete the request
1: for $w \in W$ do
2: if $w.\text{utilization} < w.\text{max_capacity}$ then $\text{available.add}(w)$
3: end for
4: do
5: $\text{choice}_1 \leftarrow X \sim \mathcal{U}(\text{available.length})$
6: $\text{choice}_2 \leftarrow X \sim \mathcal{U}(\text{available.length})$
7: while $\text{choice}_1 \neq \text{choice}_2$
8: if $\text{available}[\text{choice}_1].\text{loadsize} > \text{available}[\text{choice}_2].\text{loadsize}$ then
9: $\text{choice} \leftarrow \text{choice}_2$ else $\text{choice} \leftarrow \text{choice}_1$
9: $\text{available}[\text{choice}].\text{items_queue.put}(d)$
10: $\text{available}[\text{choice}].\text{loadsize} \leftarrow \text{available}[\text{choice}].\text{loadsize} + d.\text{size}$
return $\text{available}[\text{choice}]$

Fig. 6: Load balancing algorithm.

Figure 6 shows the load-balancing algorithm implemented in the manager, where a worker is selected by first choosing 2 different random workers from a list of available workers, and then choosing the worker with the lowest assigned load.

Different parallel patterns can be built by using this model. For instance, divide&conquer (*dq*) can be built by reusing the manager, which takes 2 roles: *divide* and *conquer*. The *divide* splits each incoming content into n segments (equal to the number of workers) and delivers them to the workers. The *workers* process the segments and deliver the results to the manager that takes the role of *conquer*, which consolidates the results into a single one. A puzzle (Φ) that includes a parallel piece is denoted as follows:

$$\Phi = [\rho_{(source,1)} \rightarrow \varphi(P_1)], [\varphi(P_1) \rightarrow P_2] \quad (4)$$

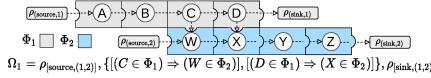


Fig. 7: Conceptual representation of a metapuzzle.

where P_2 is a regular piece, and $\varphi(P_1)$ is a parallel piece such that $\varphi(P_1) = M(3, mw) \Leftrightarrow \{w_1 \parallel w_2 \parallel w_3\}$. The execution of clones results in concurrent processing of digital products and improves the response time of that piece. Moreover, these tasks are performed implicitly and transparently. Thus, the piece performs the same functionality as the original app but improves its performance.

3.4 Modelling metapuzzles: building infrastructure-agnostic services

A puzzle is chained with other puzzles to create more complex processing services called metapuzzles (Ω). In this process, each puzzle is abstracted as a new piece; as a result, a puzzle can be coupled with other puzzles by recursively coupling pieces. Similar to a puzzle, in a metapuzzle, an arrow denotes the chaining of 2 puzzles. For example, $(P_1 \in \Phi_1) \Rightarrow (P_2 \in \Phi_2)$ means that the piece P_1 in the puzzle Φ_1 delivers results to the piece P_2 in the puzzle Φ_2 .

A metapuzzle thus is composed of the sets of sources, sinks, and puzzles, as follows:

$$\begin{aligned} \Omega = & \rho_{[\text{source},(1,\dots,n)]}, \{[(P_1 \in \Phi_1) \Rightarrow (P_1 \in \Phi_2)], \dots, \\ & [(P_a \in \Phi_x) \Rightarrow (P_b \in \Phi_y)]\}, \rho_{[\text{sink},(1,\dots,m)]} \end{aligned} \quad (5)$$

where n is the number of sources, m is the number of sinks, whereas x and y are identifiers of a puzzle (Φ), and a and b are identifiers of the pieces used to couple the puzzles. Figure 7 depicts a conceptual representation and notation of a metapuzzle integrated by 2 puzzles (Φ_1 and Φ_2). We chain puzzles Φ_1 and Φ_2 by coupling the pieces P_C and P_D of Φ_1 with P_W and P_X of Φ_2 , respectively.

Metapuzzles, puzzles, pieces, sources, and sinks are incorporated into a service mesh (Ψ). This mesh produces a repository and, from it, organizations choose the pieces required to create services to manage their digital products. It also includes access control mechanisms to only allow authorized users to access the pieces, and a manager to synchronize the message passing through the pieces. In this context, a service mesh (Ψ) is denoted, as the set of all pieces and puzzles created, as follows:

$$\Psi = [\mathbf{S}_{\text{sources}}, \Phi \wedge \Omega \wedge \mathbf{P}, \mathbf{S}_{\text{sinks}}], \quad (6)$$

where $\mathbf{S}_{\text{sources}}$ is the set of sources, $\mathbf{S}_{\text{sinks}}$ is the set of sinks, Ω is the set of metapuzzles, Φ is the set of puzzles, and \mathbf{P} is the set of pieces.

We can identify 3 features in this self-contained method: *i)* the adaptability to assimilate new stages in a lifecycle by only readjusting either the loops or sockets of a piece and by coupling it to other pieces to create a puzzle. *ii)* the re-utilization of puzzles by coupling them to other puzzles, creating a metapuzzle. And, *iii)* the automatic dataflow created through the pieces, puzzles, and metapuzzles as those structures are in charge of delivering data to the next component in the pattern.

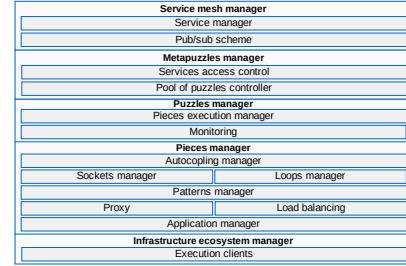


Fig. 8: PuzzleMesh architecture.

4 DESIGN PRINCIPLES AND IMPLEMENTATION OF A PROTOTYPE BASED ON PUZZLEMESH MODEL

In this section, we present the design principles for the development and implementation of a prototype based on the PuzzleMesh model described in Section 3.

In the design phase, designers define services by choosing a set of pieces/puzzles/metapuzzles from the service mesh. Also, designers can create a new piece for these apps by using a declarative scheme. The sockets and loops for each piece, as well as the coupling of the pieces, are specified by designers and PuzzleMesh produces a DAG, which is converted into a configuration file (YML). Figure 8 depicts the conceptual representation of a hierarchy architecture for managing metapuzzles, puzzles, and pieces. As can be seen, it is composed of managers for service mesh, metapuzzles, puzzles, and infrastructure ecosystem.

In the building phase, the service mesh manager, by following the hierarchy architecture, invokes the metapuzzle, puzzle, and piece managers recursively by following the configuration file created by the designer. In the deployment and orchestration phases, the service mesh manager first creates the container images of the microservices considered in the configuration file. In a second step, the managers (puzzle, metapuzzle, and piece) read the management metadata of each artifact to prepare its functionality (by adding apps, control structures such as APIs, load-balancer, and data exchange structures to the images). Moreover, during this phase, a pattern manager creates and couples the components required to implement parallel patterns inside of pieces, when this is configured to do it.

In the launching and choreographic phases, the managers recursively perform first the launching of a set of container instances by using the images previously created and then perform the auto-coupling when the managers follow the loops and sockets declared in the configuration file. A system of container instances organized in the form of a pattern is the result of this phase. In the execution and operation phases, the processing structure (including any combination of pieces, puzzles, or metapuzzles) first is exposed as a service, and later it is available for authorized end-users to operate it. The infrastructure ecosystem manager includes different clients and APIs for the communication with the infrastructure (any of edge, fog, or cloud) where the pieces are deployed.

In summary, the building of services is performed by going down through the architecture hierarchy, whereas the

execution and operational controls are established from the bottom to the top of the architecture. The service mesh manager manages the resultant services using a publication method based on a pub/sub scheme. In the PuzzleMesh prototype, *pieces* were implemented as microservices using Python programming language. Each puzzle piece implements an HTTP/REST API interface (implemented in Flask framework) for the communication between microservices through the loops and sockets.

PuzzleMesh implements an interface that executes a data preparation scheme [44] for the retrieval/delivery of the data from/to pieces. This scheme includes the LZ4 algorithm [45] to compress the data and a data deduplication algorithm for efficiency, the Information Dispersal Algorithm (IDA) [38] for reliability, and CP-ABE [46] to add confidentiality and access control to the data. The efficiency and reliability algorithms are developed in C, whereas the confidentiality algorithm is developed in Java. The managers implement I/O functions of a content delivery network called SkyCDS [43] for serverless and/or multi-cloud scenarios. The pieces are automatically converted into virtual container images and their corresponding configuration files (YML). In the PuzzleMesh prototype, the images and configuration files are delivered to the Docker engine, including Docker Compose for multi-container systems and supporting the deployment of services on Docker Swarm and Kubernetes platforms. This means PuzzleMesh delegates to the Docker container platform [47] the tasks of management and allocation of the containers and microservices created during the service construction procedure. When the instances have been deployed by the container platforms, the coupling of pieces is automatically triggered by an initiation process. In execution time, the pieces are in charge of the data preparation, data exchange, the I/O traffic management, and the task execution by using PuzzleMesh control software, such as load-balancing, coupling interfaces, and data exchanging (filesystem, memory, or network) as well as the deployment of containers (managers, orchestrator, launchers, etc.) the management of parallel patterns, and the logging system. This control software and the components of the service mesh (see Figure 8) were also developed in C/C++.

5 EXPERIMENTAL EVALUATION: A CASE STUDY BASED ON SATELLITE IMAGERY

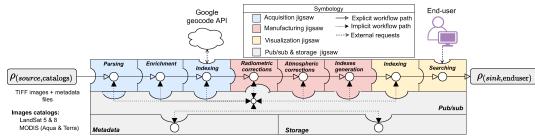


Fig. 9: Conceptual representation of the case study conducted to evaluate the PuzzleMesh prototype.

In this section, we present the experimental evaluation of the PuzzleMesh prototype, which was conducted in the form of a case study for the management and processing of satellite imagery captured by a ground station called AEM-ERIS [30].

Figure 9 depicts the evaluation scenario designed to conduct this case study. This scenario includes 3 puzzles: *Acquisition*, *Manufacturing*, and *Visualization*. Three support puzzles were also constructed: *Publish/Subscribe*, *Storage*, and *Metadata* services. The puzzles were integrated into a metapuzzle service to manage the lifecycle of satellite imagery produced by the AEM-ERIS ground station.

This metapuzzle service starts with the *Acquisition puzzle*, which invokes a piece (at the edge) that reads images from the server storing imagery. Another piece performs the parsing and indexing of the metadata of the images and the enrichment of the metadata from external sources (e.g., Google Geocoding API).

The *Manufacturing puzzle* (at the fog) builds derivative products from the satellite imagery by executing a pipeline of 3 pieces. The first piece executes an algorithm for the radiometric corrections of the bands of each image (16/32 sub-images). The second piece executes an algorithm for the atmospheric correction of the images. The last piece executes an app for building derivative earth observation products from corrected images, which produces 8 environmental indexes: normalized difference vegetation index (NDVI), enhanced vegetation index (EVI), soil-adjusted vegetation index (MSAVI), normalized difference moisture index (NDMI), normalized burn ratio (NBR), NBR2, and normalized difference snow index (NDSI) [48].

The *Visualization puzzle* is a geoportal service where the images are indexed to make them available to end-users.

The *Acquisition puzzle* sends the imagery and processed metadata to the storage, pub/sub, and metadata puzzles (at the cloud), which are also used by *Manufacturing* and *Visualization* puzzles to deliver/retrieve data and metadata.

This case study is conducted in 4 phases. In the first one, different load-balancers for parallel patterns were evaluated. In the second one, the 3 puzzles (acquisition, manufacturing, and visualization) were evaluated in an isolated manner. In the third phase, different metapuzzles were built by performing combinations of the 3 puzzles. In the second phase, the metapuzzle including the 3 puzzles was evaluated in a direct comparison with state-of-the-art tools Pegasus [20], Makeflow [35], and DagOnStar [19], and the CI/CD tool Jenkins [33].

5.1 Environment of experimentation

Two clusters of computers and virtual machines were configured to conduct the experimental evaluation (see Table 2). Four machines integrate the first cluster located in Cinvestav Tamaulipas. Four virtual machines integrate the second cluster deployed on Amazon EC2 (West region).

Phase 1 and 2 of the experimental evaluation were performed by using the first cluster located at Cinvestav Tamaulipas, and the third phase was performed by using both clusters. Each puzzle (acquisition, manufacturing, and visualization) was deployed and evaluated on different computers (either virtual machines or containers). In the case of the Cinvestav's cluster, Compute 1 was used as a storage node, whereas Computes 2, 3, and 4 were used to deploy the puzzles, respectively.

The dataset used in this case study considers data that AEM-ERIS captures from platforms such as MODIS (Aqua

TABLE 2: Characteristics of the infrastructure.

Label	Location	Cores	RAM (GB)	Storage
Compute 1	Cinvestav	12	64	3.2 TB
Compute 2,3, & 4	Tamaulipas	12	64	2.7 TB
Cloud 1, 2, 3, & 4	Amazon EC2 (West region)	16	32	600 GB

TABLE 3: Characteristics of the satellite imagery.

Catalog	Total size (GB)	Number of products	Images size (MB)	Acquisition date
Terra	131.11	13965	185.35	Jul 2008 – Mar 2011
Aqua	172.15	1011	174.53	Dec 2010 – Sep 2013
Landsat5	448.24	1662	275.67	Jul 2007 – March 2016
Landsat8	38	40	911.06	November 2011 – May 2020

and Terra) and Landsat (Landsat 5), which are managed as a repository of 3 catalogs. Furthermore, a crawler piece recovered data from Landsat 8 imagery acquired from the USGS Earth Explorer [49] and was also added as a fourth catalog of the repository. Table 3 shows a brief description of the main characteristics of the images.

5.2 Phase 1: Evaluating data load-balancing in parallel patterns

We perform a direct comparison of the two-choices algorithm implemented in PuzzleMesh with Random and Round-Robin algorithms used in the literature to distribute workload between workers [38]. We evaluated the algorithms distributing the workload to a pattern of 12 workers. We measured the workload as the amount of data and tasks sent to each worker, managed by a piece by each algorithm. To perform this experiment, we distribute a set of 68 GB (575 heterogeneous files of satellite data) to a piece executing 12 parallel workers. We calculated the ideal workload for a worker is calculated as $|F|/w$, where $|F|$ is the total workload and w is the number of workers in the pattern. Thus, the ideal workload (IW) with 12 workers for this experiment is 3.16 GB per worker.

We also calculated the percentage of error produced by each algorithm compared with this ideal workload. The percentage of error thus measures the effectiveness of a load-balancing algorithm. Figure 10 shows on the vertical axis the percentage of error observed for each worker (horizontal worker) by using the 3 load-balancers considered in this evaluation. As it can be observed, two-choices produced a workload close to the ideal, with a percentage of error of no more than 1.7%. In contrast, the Round-Robin algorithm, which is the common algorithm used in industry, produced an error of 23.33% because this algorithm relies on that all files will be homogeneous to create a IW . However, this is not the case of realistic scenarios where it is expected to process sets of heterogeneous files. Moreover, the more the files in a data source, the closer to zero the error percentage of the two-choices algorithm.

5.3 Phase 2: analyzing the performance of implicit parallel patterns

In this phase, we evaluated the performance, in an isolated manner, of the tree puzzle services built with PuzzleMesh (*acquisition*, *manufacturing*, and *visualization*). The purpose of these experiments is to show how the patterns of the PuzzleMesh model can reduce the impact produced by the costliest pieces of the services on end-user experience. In this

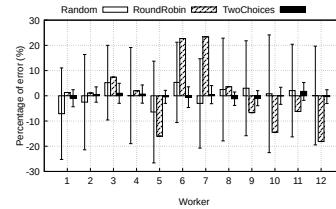


Fig. 10: Percentage of error of the workload assigned to each worker using 3 different load-balancing algorithms.

phase, each puzzle was deployed on a different physical machine in the Cinvestav container cluster (fog), which includes 12 physical cores per machine. The metrics for the evaluation were the *service times* and *throughput* of the pieces and puzzles. Each piece deploys 12 parallel workers running on 12 containers. For example, the acquisition and manufacturing stages deployed 36 containers (12 workers \times 3 pieces), whereas the visualization stage deployed 24 containers (12 workers \times 2 pieces). Each virtual container deploys a worker limited to one physical core.

Figure 11a shows, in the vertical-left axis, the performance of the service time spent by the pieces of the *acquisition* puzzle to parse, enrich, and index the metadata of the 4639 products (1.01 TB of imagery and 17.85 MB of metadata files) is processed by using parallel patterns including a different number of workers (horizontal axis) per each piece in the puzzle. The vertical-right axis shows the throughput of the solution.

As expected, the parallel pattern and the load-balancing on the containers significantly reduced the service time of the puzzle service, which increased the number of files processed per time unit. As can be seen, the more workers in the pattern, the more parallelism and the fairest load-balancing and work distribution, the more the improvement in the puzzle service performance.

Specifically, the *M/W* pattern executing 12 parallel workers (clones of a piece equal to the number of physical cores in the fog server) produces an improvement in the response time of 80.82% compared with running a traditional server (only one worker). We observed that the metric that imposes this restrictive behavior is the number of cores in the physical machine where the containers of the workers are deployed on. In the case of enabling virtual cores in the servers, an improvement could still be observed, but with a lesser impact than that one observed when using physical cores. It was observed thus that is suitable to choose as many workers (clones) as physical cores available in the physical machine when processing large size contents (e.g., images). Nevertheless, it also was observed that it is suitable to use as many workers as the number of virtual plus physical cores when processing small data portions (e.g., metadata, documents, or PDFs). The experimental evaluation revealed that overhead produced by PuzzleMesh was not significant to affect the performance of the pieces. This overhead represents less than 1% (in median) of the service time yield by each pattern evaluated in these studies. This overhead produced by PuzzleMesh was calculated by

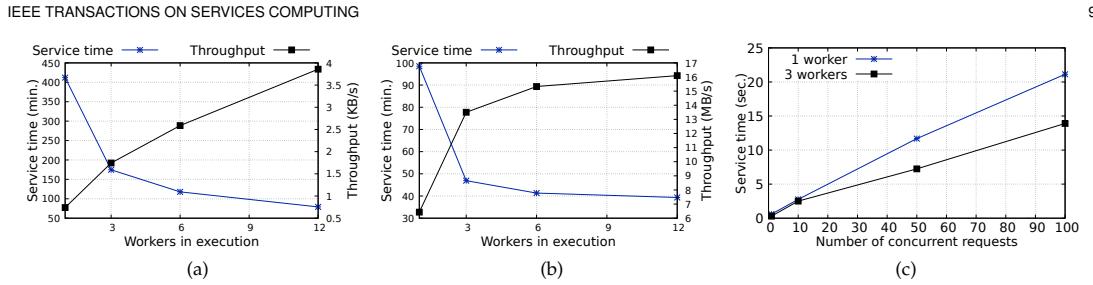


Fig. 11: Performance evaluation of (a) acquisition, (b) manufacturing, and (c) visualization puzzle services.

the sum of times of the load-balancing, and tasks execution management. Note that the overhead is absorbed by the performance gain yield by the implicit parallelism and the load-balancing.

Figure 11b shows, on the vertical axis, the service time of the pre-processing pieces of *manufacturing* when a *M/W* pattern correcting 50 digital products is deployed and the number of workers is increased (horizontal axis). This time considers the correction of each band of a satellite image and the creation of a new version of the image (a corrected image). As it can be seen, the parallel pattern implemented in *manufacturing* puzzle service yields the previously observed impact on the performance of the service. As was expected, the service time was reduced as more containers were running in parallel. For instance, for 12 parallel workers, an improvement in the service time of 60.05% was observed compared with the execution of only one container.

We have also tested the performance of a *visualization* puzzle service by using a *M/W* parallel pattern. In this case, all the pieces (microservices) of *visualization* were encapsulated into a puzzle service that was cloned and replicated 3 times to create a metapuzzle. Figure 11c shows the service time, vertical axis, produced when *visualization* puzzle service and when it was built by using a metapuzzle service by executing a pattern of parallel puzzles (three clones of *visualization* puzzle) to serve requests from a different number of concurrent queries (horizontal axis). As it can be seen, the configuration including a pattern of 3 replicas of the *visualization* puzzle reduced the service time in proportion with the increment of concurrent requests compared with a traditional solution based on a single *visualization*, which is consistent with the distribution of queries in a load-balancing manner among the puzzles of the final service. This type of metapuzzles can be also used to create, not only available but also reliable infrastructure-agnostic services. As it can be noted, the cloning of components applies to pieces, puzzles, and metapuzzles to create patterns at different levels, and that the organizations can set up the number of replicas in the patterns depending on the workloads and infrastructure available for the deployment of a given solution. In this context, the PuzzleMesh model automatically and transparently adapts the dataflows to these parameters.

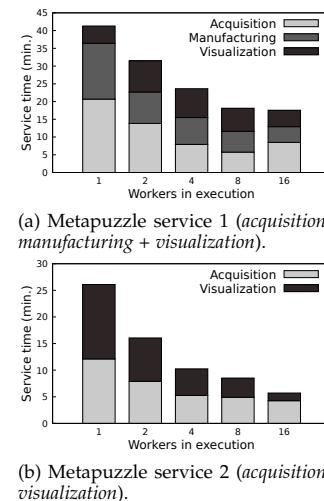


Fig. 12: Evaluation of the metapuzzles created by combining puzzles.

5.4 Phase 3: analyzing the performance of the comprehensive services

To conduct the third evaluation phase, we combined *acquisition*, *manufacturing*, and *visualization* puzzles to create 2 metapuzzles. In the first one, *acquisition* acquires digital products, *manufacturing* process these digital products to create corrected versions of the satellite imagery, which are exhibited by *visualization* service. In the second one, only *acquisition* and *visualization* puzzles were considered in the final service. The goal of this evaluation is to show the feasibility of building comprehensive services by recursively combining pieces, puzzles to create metapuzzles.

The 2 metapuzzle services created for this evaluation phase were tested by processing and managing 50 satellite images, each experiment was performed 31 times to achieve a median value of the evaluated metrics (service times).

Figure 12 shows the service times (vertical axis) produced by services 1 (see Figure 12a) and 2 (see Figure 12b) when increasing the number of containers (pieces) executed in parallel (horizontal axis). In this experiment, each service (*acquisition*, *manufacturing*, and *visualization*) is executed in patterns that include the same number of workers (puzzles).

For example, column 2 in Figure 12b shows that *acquisition* puzzle was executed by using a pattern including 2 clones of the original puzzle (*Acquisition*), which were deployed on 2 different containers running in parallel. This was also applied to the *visualization* puzzle service.

As can be seen in Figures 12a and 12b, the service time is reduced for each metapuzzle (and for each puzzle service) when increasing the number of containers running in parallel. For instance, for service 2, the patterns running 16 containers per service (Column 5 in Figure 12b) produces an improvement in the service time of 78.15% compared with a service without parallel patterns (see column 1 in Figure 12b), whereas for service 1 the percentage of gain is of 57.50% when comparing the version running 16 parallel workers with the version without parallelism.

As expected, the number of clones can be greater than the cores of the servers where the patterns are deployed when processing small data pieces, which is the case of service 2 that only processes the metadata of the imagery. However, it is recommended to launch as many containers as cores in a server when processing large data pieces, which is the case of service 1 in Figure 12a where *manufacturing* performs radiometric and atmospheric corrections to satellite images.

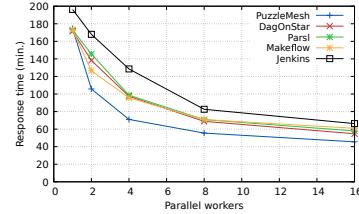
5.5 Phase 4: Direct comparison with state-of-the-art solutions

In the last phase of the experimentation, the performance of PuzzleMesh services was compared with solutions built by using state-of-the-art frameworks. In this experiment, a service including the chaining of the Acquisition, Manufacturing, and Visualization services was built by using the following frameworks: *i*) *DagOnStar* [19] is a workflow engine that produces implicit task parallelism; *ii*) *Parsl* [18] is a workflow engine that includes Slurm [11] for resources management; *iii*) *Makeflow* [35], is a workflow engine that includes HTCondor [10] for resources management; *iv*) *Jenkins* [33], a platform to create CI/CD pipelines; and *v*) *PuzzleMesh* solution.

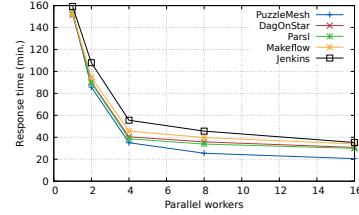
Two types of experiments were performed in this evaluation phase. In the first one, we executed the chain of services on a local cluster located at Cinvestav Tamaulipas. In the second one, we executed it on virtualized resources at the Amazon EC2 cloud (see Table 2). To enable the movement of data between services in the cluster, for each solution implemented, the *Storage* service was added as a processing stage to the workflows created with *DagOnStar*, *Parsl*, *Jenkins*, and *Makeflow*.

These services were used for creating a pipeline of applications managed as a service, where the acquisition stage automatically extracts the 40 images from the data source, which were delivered to the manufacturing stage, where these images were used to create derivative products. These products were delivered to the visualization stage, where they are indexed in a database for end-user consumption.

In *DagOnStar*, *Makeflow*, *Jenkins*, and *Parsl*, the management of the data is performed through SCP. This means, that an SCP user to enable data movement between stages. In *PuzzleMesh*, the data movement is transparently performed by the Pub/Sub and Storage pieces, which implement a native CDN (SkyCDS [43]). For *DagOnStar*, *Makeflow*, *Jenkins*,



(a) Local cluster.



(b) Cloud.

Fig. 13: Response time observed to process the LandSat 8 dataset in *a*) a local cluster and *b*) the cloud.

and *Parsl* parallelism was managed using a threads-model available in each framework. The number of threads was varied using the same number of cores for each solution evaluated. Each service was installed on the same infrastructure, but not executed at the same time. Thus, *PuzzleMesh* and the state-of-the-art solutions were deployed and executed in the same environment, processing the same data source, performing the very same experiments, and capturing the same metrics at the same periods.

The metric evaluated in these experiments was the *response time*, which represents the end-user experience. Again, each experiment was performed 31 times to achieve a median. Figure 13a shows the results obtained from the execution of the services in the local resources at Cinvestav, whereas Figure 13b shows the results observed at the cloud. Both Figures show in the vertical axis the response time in minutes by using a different number of parallel workers for each piece of each puzzle in the case of *PuzzleMesh* and for a different number of threads in the case of the rest of the solutions built by using state-of-the-art frameworks (horizontal axis).

Figure 13a shows that *PuzzleMesh* processed the data in 45.54 minutes by using 16 workers in the patterns when the service was deployed on local resources (edge and fog), whereas *DagOnStar*, *Parsl*, *Makeflow*, and *Jenkins* performed the same operation (with 16 parallel threads) in 54.75, 57.94, 60.94, and 66.26 minutes, respectively. That means an improvement in the response time of 16.81%, 21.39%, 25.26%, and 31.26% of *PuzzleMesh* in comparison with *DagOnStar*, *Parsl*, *Makeflow*, and *Jenkins* respectively.

Similar behavior is observed when the service was executed at the cloud by using each tool. Figure 13b shows that the response time of the tools is reduced compared with the local version, and the percentage of gain of *PuzzleMesh* in comparison with the other tools is increased. This can be explained due to the changes in the infrastructure, for

example, a higher number of available CPUs in the cloud instances. PuzzleMesh processed the data in 20.54 minutes by using 16 workers in the patterns when the workflow is deployed on local resources, whereas DagOnStar, Parsl, Makeflow, and Jenkins performed the same operation (with 16 parallel threads) in 30.75, 29.94, 33.94, and 35.26 minutes, respectively. This means an improvement in the response time of 33.18%, 31.38%, 39.47%, and 41.73% of PuzzleMesh compared with DagOnStar, Parsl, Makeflow, and Jenkins, respectively. This confirms our initial observation: the more the available resources, the more the improvement produced by the service built by PuzzleMesh model.

Note that these results denote that the performance of PuzzleMesh is competitive with tools from state-of-the-art under the specifications of the case study conducted in the experimental evaluation. The performance of these solutions could be changed when different configurations and infrastructure features are fine-tuned to achieve the bests feasible configurations for each solution.

The case study and the comparison with state-of-the-art solutions revealed the feasibility of PuzzleMesh to build comprehensive infrastructure-agnostic solutions abstracted as puzzles and metapuzzles.

6 CONCLUSIONS AND FUTURE WORK

This paper presented the design and implementation of a model called *PuzzleMesh* for creating infrastructure-agnostic services. This construction model is based on a puzzle metaphor and a self-contained building method that integrates, parallel pattern managers, load-balancers, launchers, and I/O managers with an app to create a single cohesive software artifact called “piece”. This enables designers to create autonomous software pieces, which, immediately, can be recursively coupled to other pieces to create processing structures in the form of puzzles and/or metapuzzles.

The experimental evaluation of the PuzzleMesh considered case studies based on the building of services for managing satellite imagery lifecycles. This evaluation revealed the efficiency of data processing by using self-contained pieces, puzzles, and metapuzzles. A quantitative comparison son with state-of-the-art solutions showed the performance efficiency of this model. The impact of self-contained in the processing of large volumes of data was described in terms of the reduction of bottlenecks and waiting times. This improved the service and response times as well as the processing throughput of the services, which is key in decision-making processes. The deploying of multiple combinations of puzzles and metapuzzles services on both fog and cloud infrastructures during the experimental evaluation showed the reusability and portability of the PuzzleMesh model.

A qualitative assessment revealed the key characteristics that enable PuzzleMesh to build processing structures as services in real-world scenarios and the main difference with similar frameworks available in the state-of-the-art. Ongoing work is the implementation of services for real-world use cases such as, but not limited to, environmental workflows for extreme weather events early detection [50]. We are considering as future work an adaptive continuous dataflow scheme for mitigating, in execution time, side effects of bottlenecks when facing up changes in workload

and/or the infrastructure (e.g., in cloud-to-cloud migration or *C2C*), which is not studied in this paper. Adaptive maintenance schemes for microservice architecture built with our model also are currently under study for providing services with implicit security and fault-tolerance schemes.

ACKNOWLEDGMENTS

This work has been partially supported by the FORDECYT-PRONACES project 41756 “Plataforma tecnológica para la gestión, aseguramiento, intercambio y preservación de grandes volúmenes de datos en salud y construcción de un repositorio nacional de servicios de análisis de datos de salud”; the Spanish Ministry of Science and Innovation Project “New Data Intensive Computing Methods for High-End and Edge Computing Platforms (DECIDE)” (PID2019-107858GB-I00); and by the Ministry of Education and Science of the Russian Federation (075-15-2020-788).

REFERENCES

- [1] G. Vazquez, J. Gonzalez, V. Sosa, M. Morales, and J. Perez, “Cloud-chain: A novel distribution model for digital products based on supply chain principles,” *IJIM*, vol. 39, pp. 90–103, 2018.
- [2] R. Lobato, *Netflix nations: The geography of digital distribution*. NYU Press, 2019.
- [3] Daki, E. Hannani, Aqqal, Haidine, and Dahbi, “Big data management in smart grid: concepts, requirements and implementation,” *Journal of Big Data*, vol. 4, no. 1, pp. 1–19, 2017.
- [4] N. Tavares, W. Hasselbring, T. Weber, and D. Kranzlmüller, “Designing a generic research data infrastructure architecture with continuous software engineering,” in *SE-WS*, 2018, pp. 85–88.
- [5] M. Rodriguez and R. Buyya, “A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments,” *CCPE*, vol. 29, no. 8, p. e4041, 2017.
- [6] F. Marozzo, D. Talia, and P. Trunfio, “A workflow management system for scalable data mining on clouds,” *IEEE Transactions on Services Computing*, vol. 11, no. 3, pp. 480–492, 2018.
- [7] M. Santiago, J. Gonzalez *et al.*, “A gearbox model for processing large volumes of data by using pipeline systems encapsulated into virtual containers,” *FIGS*, vol. 106, pp. 304–319, 2020.
- [8] Reyes, Gonzalez, Sosa, Carretero, and Garcia, “Kulla, a container-centric construction model for building infrastructure-agnostic distributed and parallel applications,” *JSS*, p. 110665, 2020.
- [9] Holm *et al.*, “Cloudflow-an infrastructure for engineering workflows in the cloud,” in *UBICOMM 2016*, vol. 10, 2016, pp. 158–165.
- [10] E. Fajardo, J. Dost, B. Holzman, T. Tannenbaum, J. Letts, A. Tiradani, B. Bockelman, J. Frey, and D. Mason, “How much higher can htcondor fly?” in *JPCS*, vol. 664, 2015, p. 062014.
- [11] B. Christiansen, M. Garey, and I. Hartung, “Sturm overview,” 2017.
- [12] D. Sánchez *et al.*, “From the edge to the cloud: A continuous delivery and preparation model for processing big iot data,” *SIMPAT*, vol. 105, p. 102136, 2020.
- [13] A. Hosny, P. Vera-Licona, R. Laubenbacher, and T. Favre, “Alggorun: a docker-based packaging system for platform-agnostic implemented algorithms,” *Bioinformatics*, vol. 32, no. 15, pp. 2396–2398, 2016.
- [14] Bogomolov *et al.*, “Authorship attribution of source code: A language-agnostic approach and applicability in software engineering,” in *29th ESEC/FSE*, 2021, pp. 932–944.
- [15] J. Opara-Martins *et al.*, “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective,” *Journal of Cloud Computing*, vol. 5, no. 1, pp. 1–18, 2016.
- [16] S. Iserete, J. Prades, C. Reaño, and F. Silla, “Increasing the performance of data centers by combining remote gpu virtualization with slurm,” in *CCGrid*. IEEE, 2016, pp. 98–101.
- [17] Taufer *et al.*, “Scheduling dag-based workflows on single cloud instances: High-performance and cost effectiveness with a static scheduler,” *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 19–31, 2017.
- [18] Y. Babuji *et al.*, “Parsl: Pervasive parallel programming in python,” in *28th HPDC*, 2019, pp. 25–36.

- [19] D. D. Sánchez-Gallegos *et al.*, "An efficient pattern-based approach for workflow supporting large-scale science: The dragonstar experience," *FGCS*, vol. 122, pp. 187–203, 2021.
- [20] E. Deelman *et al.*, "The evolution of the pegasus workflow management software," *CISE*, vol. 21, no. 4, pp. 22–36, 2019.
- [21] V. der Aalst *et al.*, "Processes meet big data: Connecting data science with process science," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 810–819, 2015.
- [22] Ardagna, Bellandi, Bezzi, Ceravolo, Damiani, and Hebert, "Model-based big data analytics-as-a-service: take big data to the next level," *IEEE Transactions on Services Computing*, 2018.
- [23] Mohan and Kangasharju, "Edge-fog cloud: a distributed cloud for internet of things computations," in *8th CloT*. IEEE, 2016, pp. 1–6.
- [24] D. Sánchez *et al.*, "On the continuous processing of health data in edge-fog-cloud computing by using micro/nanoservice composition," *IEEE Access*, vol. 8, pp. 120255–120281, 2020.
- [25] J. Opara-Martins, "Taxonomy of cloud lock-in challenges," in *Mobile Computing-Technology and Applications*. IntechOpen, 2018.
- [26] N. Haili and J. Altmann, "Evaluating investments in portability and interoperability between software service platforms," *FIGS*, vol. 78, pp. 224–241, 2018.
- [27] P. Nguyen and K. Nahrstedt, "Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows," in *2017 IEEE ICAC*. IEEE, 2017, pp. 187–196.
- [28] M. Paschali *et al.*, "Reusability of open source software across domains: A case study," *JSS*, vol. 134, pp. 211–227, 2017.
- [29] S. Battula *et al.*, "An efficient resource monitoring service for fog computing environments," *IEEE Transactions on Services Computing*, 2019.
- [30] ECOSUR, "Eris," July 28, 2020. [Online]. Available: <https://www.ecosur.mx/tag/antena-eris/>
- [31] M. Kasztelnik *et al.*, "Support for taverna workflows in the vph-share cloud platform," *Computer Methods and Programs in Biomedicine*, vol. 146, pp. 37–46, 2017.
- [32] J. Conejero *et al.*, "Task-based programming in compss to converge from hpc to big data," *IJHPCA*, vol. 32, no. 1, pp. 45–60, 2018.
- [33] J.-M. Belmont, *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI*. Packt Publishing Ltd, 2018.
- [34] Madduri *et al.*, "The globus galaxies platform: delivering science gateways as a service," *CCPE*, vol. 27, no. 16, pp. 4344–4360, 2015.
- [35] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makewflow, work queue, and docker," in *HPDC'15*, 2015, pp. 31–38.
- [36] "Istio," December, 2020. [Online]. Available: <https://istio.io/>
- [37] W. Caban, "The openshift architecture," in *Architecting and Operating OpenShift Clusters*. Springer, 2019, pp. 1–29.
- [38] Morales *et al.*, "A data distribution service for cloud and containerized storage based on information dispersal," in *SOSE*. IEEE, 2018, pp. 86–95.
- [39] M. Phillips, "International data-sharing norms: from the oecd to the general data protection regulation (gdpr)," *Human genetics*, vol. 137, no. 8, pp. 575–582, 2018.
- [40] Y. Zhang, R. Deng, X. Liu, and D. Zheng, "Outsourcing service fair payment based on blockchain and its applications in cloud computing," *IEEE Transactions on Services Computing*, 2018.
- [41] D. B. Rawat *et al.*, "Cybersecurity in big data era: From securing big data to data-driven security," *IEEE Transactions on Services Computing*, 2019.
- [42] B. Barker, "Message passing interface (mpi)," in *Workshop: High Performance Computing on Stampede*, vol. 262, 2015.
- [43] J. Gonzalez, J. Perez, V. Sosa, L. Sanchez, and B. Bergua, "Skycds: A resilient content delivery service based on diversified cloud storage," *SIMPAT*, vol. 54, pp. 64–85, 2015.
- [44] D. Carrizales *et al.*, "A data preparation approach for cloud storage based on containerized parallel patterns," in *IDCS 2019*. Springer, 2019, pp. 478–490.
- [45] LZ4, "Lz4: extremely fast compression algorithm." [Online]. Available: <https://lz4.github.io/lz4/>
- [46] Ding *et al.*, "A novel efficient pairing-free cp-abe based on elliptic curve cryptography for iot," *IEEE Access*, vol. 6, pp. 27336–27345, 2018.
- [47] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sept 2014.
- [48] E. Vermote *et al.*, "Preliminary analysis of the performance of the landsat 8/oli land surface reflectance product," *RSE*, vol. 185, pp. 46–56, 2016.
- [49] USGS, "Earthexplorer - usgs," July 31, 2020. [Online]. Available: <https://earthexplorer.usgs.gov/>
- [50] R. Montella, D. Di Luccio, A. Ciaramella, and I. Foster, "Stormseeker: A machine-learning-based mediterranean storm tracer," in *IDCS 2019*. Springer, 2019, pp. 444–456.



Dante D. Sánchez-Gallegos is a PhD student at Cinvestav Tamaulipas, Mexico. He received a B.E. degree in IT engineering from the Polytechnic University of Victoria, Tamaulipas, Mexico, in 2016. In 2019, he received a master in sciences degree from the Cinvestav Tamaulipas, Mexico. His research areas of interest are processing workflows, distributed systems, and cloud computing.



J. L. González-Compean received his Ph.D. in Computer architecture from UPC Universitat Politecnica de Catalunya, Barcelona (2009). He was a visiting professor at Universidad Carlos III de Madrid, Spain and researcher at Cinvestav, Tamaulipas, Mexico. His research lines are Cloud-Based Storage systems, federated storage networks, and storage virtualization.



Jesus Carretero is a senior full time professor-researcher at Carlos III University. His research lines are high-performance computing and cloud computing, parallel and distributed systems, and real-time systems. He is leader of research group ARCOS at Carlos III University and has published 17 books, has been involved in 52 research projects, and he has written more than 200 journal and congress articles.



Heidi Marin-Castro received the Ph.D. degree in Computer Science from Information Technology Lab Cinvestav, Mexico in 2014. Professor researcher at Engineering and Sciences Faculty in the Autonomous University of Tamaulipas, Mexico. Her research areas include Web Data Management, Databases, Data Mining and Information Retrieval.



Andrei Tchernykh is full professor in computer science at CICESE Research Center, Ensenada, Baja California, Mexico, and chairing the Parallel Computing Laboratory since 1995. He graduated from the Sevastopol Technical University in 1975. He received his Ph.D. degree from the Institute of Precision Mechanics and Computer Engineering (ITM&BT) of the RAS in 1986. He is a founding member of the Mexican Supercomputer Society.



Raffele Montella is an assistant professor (tenure) in Computer Science at Department of Science and Technologies, University of Naples Parthenope, Italy since 2005. PhD in Marine Science and Engineering at the University of Naples Federico II. The research main topics are focused on tools for high-performance computing, grid, cloud and GPUs with apps in computational environmental science and embedded/mobile computing and IoT.

B

Ejemplo de un archivo de configuración de PuzzleMesh

A continuación, se presenta un ejemplo de un archivo de configuración utilizado en PuzzleMesh para crear un servicio agnóstico de la infraestructura.

```
1 [BB]
2 name = Anonimizacion
3 command = python3 /code/process_dir.py --input @I --outfolder '@D' --save dicom
4 image = ddomizzi/cleaner:header
5 [END]
6
7 [BB]
8 name = ToPNG
9 command = python3 /code/dicom2rgb.py @I @D@L
10 image = ddomizzi/dicomtorgb:v1
11 [END]
12
13 [BB]
```

```
14 name = DetectorPulmon
15 command = python3 /code/detectorPulmones.py @I @D/@L
16 image = ddomizzi/deteccion:pulmon
17 [END]
18
19 [PATTERN]
20 name = Anonimizacionpattern
21 task = Anonimizacion
22 pattern = MW
23 workers = 2
24 loadbalancer = TC:DL
25 [END]
26
27 [PATTERN]
28 name = ToPNGpattern
29 task = ToPNG
30 pattern = MW
31 workers = 2
32 loadbalancer = TC:DL
33 [END]
34
35 [PATTERN]
36 name = DetectorPulmonpattern
37 task = DetectorPulmon
38 pattern = MW
39 workers = 2
40 loadbalancer = TC:DL
41 [END]
42
43 [STAGE]
44 name = stage_Anonimizacion
45 source = @PWD/DemoMiercoles/catalogs/P1
46 sink = stage_ToPNG
47 transformation = Anonimizacionpattern
48 [END]
49
50 [STAGE]
51 name = stage_ToPNG
52 source = stage_Anonimizacion
```

```
53 sink = stage_DetectorPulmon
54 transformation = ToPNGpattern
55 [END]
56
57 [STAGE]
58 name = stage_DetectorPulmon
59 source = stage_ToPNG
60 sink =
61 transformation = DetectorPulmonpattern
62 [END]
63
64 [NFR]
65 name = Manager/worker
66 [END]
67
68 [NFR]
69 name = Deduplication
70 [END]
71
72 [NFR]
73 name = Compression
74 [END]
75
76 [NFR]
77 name = AES4SEC
78 [END]
79
80 [NFR]
81 name = IDA
82 [END]
83
84 [WORKFLOW]
85 name = DemoMiercoles
86 stages = stage_Anonimizacion stage_ToPNG stage_DetectorPulmon
87 catalogs = P1:5981f826c1232348063dfc86a2cf6081909b7215367992de52d8efc073ec3af4
88 [END]
```


Bibliografía

- [1] 02DCE (2022). Software engineering | classification of software requirements. <https://www.geeksforgeeks.org/software-engineering-classification-of-software-requirements/>.
- [2] Abbas, A. and Khan, S. U. (2014). A review on the state-of-the-art privacy-preserving approaches in the e-health clouds. *IEEE journal of Biomedical and health informatics*, 18(4):1431–1441.
- [3] Adadi, A. (2021). A survey on data-efficient algorithms in big data era. *Journal of Big Data*, 8(1):24.
- [4] Adler, M., Chakrabarti, S., Mitzenmacher, M., and Rasmussen, L. (1995). Parallel randomized load balancing. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 238–247.
- [5] Albrecht, M., Donnelly, P., Bui, P., and Thain, D. (2012). Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, pages 1–13.
- [6] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2017). Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*.
- [7] Almarabeh, T., Majdalawi, Y. K., and Mohammad, H. (2016). Cloud computing of e-government. *Communications and Network*, 8(01):1–8.
- [8] Alsaadi, A., Ward, L., Merzky, A., Chard, K., Foster, I., Jha, S., and Turilli, M. (2022). Radical-

- pilot and parsl: Executing heterogeneous workflows on hpc platforms. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 27–34. IEEE.
- [9] Alyas, T., Ali, S., Khan, H. U., Samad, A., Alissa, K., and Saleem, M. A. (2022). Container performance and vulnerability management for container security using docker engine. *Security and Communication Networks*, 2022.
- [10] Amazon, E. (2022). Amazon ec2. <https://aws.amazon.com/es/ec2>.
- [11] Ardagna, Bellandi, Bezzi, Ceravolo, Damiani, and Hebert (2018). Model-based big data analytics-as-a-service: take big data to the next level. *IEEE Transactions on Services Computing*.
- [12] Armenise, V. (2015). Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery. In *Proceedings of the Third International Workshop on Release Engineering*, pages 24–27. IEEE.
- [13] Ascione, I., Giunta, G., Mariani, P., Montella, R., and Riccio, A. (2006). A grid computing based virtual laboratory for environmental simulations. In *European Conference on Parallel Processing*, pages 1085–1094. Springer.
- [14] Avram, M.-G. (2014). Advantages and challenges of adopting cloud computing from an enterprise perspective. *Procedia Technology*, 12:529–534.
- [15] Azhari, E.-E. M., Hatta, M. M. M., Htike, Z. Z., and Win, S. L. (2014). Tumor detection in medical imaging: a survey. *International Journal of Advanced Information Technology*, 4(1):21.
- [16] Babuji, Y. et al. (2019). Parsl: Pervasive parallel programming in python. In *28th HPDC*, pages 25–36.
- [17] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer.

- [18] Bao, L., Wu, C., Bu, X., Ren, N., and Shen, M. (2019). Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129.
- [19] Barclay, K. and Savage, J. (2010). *Groovy programming: an introduction for Java developers*. Elsevier.
- [20] Barika, M., Garg, S., Zomaya, A. Y., Wang, L., Moorsel, A. V., and Ranjan, R. (2019). Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. *ACM Computing Surveys (CSUR)*, 52(5):1–41.
- [21] Barker, A. and Van Hemert, J. (2007). Scientific workflow: A survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics*, pages 746–753. Springer.
- [22] Barker, B. (2015). Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, volume 262.
- [23] Barseghian, D., Altintas, I., Jones, M. B., Crawl, D., Potter, N., Gallagher, J., Cornillon, P., Schildhauer, M., Borer, E. T., Seabloom, E. W., et al. (2010). Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50.
- [24] Beard, J. C., Li, P., and Chamberlain, R. D. (2015). Raftlib: A c++ template library for high performance stream parallel processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 96–105.
- [25] Beard, J. C., Li, P., and Chamberlain, R. D. (2017). Raftlib: A c++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404.

- [26] Belmont, J.-M. (2018). *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI*. Packt Publishing Ltd.
- [27] Belqasmi, F., Glitho, R., and Fu, C. (2011). Restful web services for service provisioning in next-generation networks: a survey. *IEEE Communications Magazine*, 49(12):66–73.
- [28] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.
- [29] Bhalerao, A. and Pawar, A. (2017). A survey: On data deduplication for efficiently utilizing cloud storage for big data backups. In *2017 international conference on trends in electronics and informatics (ICEL)*, pages 933–938. IEEE.
- [30] Bogomolov et al. (2021). Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In *29th ESEC/FSE*, pages 932–944.
- [31] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16.
- [32] Brigade (2022). Brigade. <https://docs.brigade.sh/>.
- [33] Bux, M. and Leser, U. (2013). Parallelization in scientific workflow management systems. *arXiv preprint arXiv:1303.7195*.
- [34] Buyya, R., Pathan, M., and Vakali, A. (2008). *Content delivery networks*, volume 9. Springer Science & Business Media.
- [35] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616.

- [36] Caban, W. (2019). The openshift architecture. In *Architecting and Operating OpenShift Clusters*, pages 1–29. Springer.
- [37] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).
- [38] Carrizales, D. et al. (2019). A data preparation approach for cloud storage based on containerized parallel patterns. In *IDCS 2019*, pages 478–490. Springer.
- [39] Carrizales-Espinoza, D., Sanchez-Gallegos, D. D., Gonzalez-Compean, J., and Carretero, J. (2022). Fedflow: A federated platform to build secure sharing and synchronization services for health dataflows. *Computing*, pages 1–19.
- [40] Casado, R. and Younas, M. (2015). Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091.
- [41] Chadha, M., John, J., and Gerndt, M. (2020). Extending slurm for dynamic resource-aware adaptive batch scheduling. *arXiv preprint arXiv:2009.08289*.
- [42] Chard, R., Babuji, Y., Li, Z., Skluzacek, T., Woodard, A., Blaiszik, B., Foster, I., and Chard, K. (2020). Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 65–76.
- [43] Christiansen, B., Garey, M., and Hartung, I. (2017). Slurm overview.
- [44] Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (2012). *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media.
- [45] Cid-Fuentes, J. Á., Alvarez, P., Amela, R., Ishii, K., Morizawa, R. K., and Badia, R. M. (2020). Efficient development of high performance data analytics in python. *Future Generation Computer Systems*, 111:570–581.

- [46] Cieslik, M. and Mura, C. (2014). PaPy: Parallel and distributed data-processing pipelines in Python. *arXiv preprint*.
- [47] Clark, K., Vendt, B., Smith, K., Freymann, J., Kirby, J., Koppel, P., Moore, S., Phillips, S., Maffitt, D., Pringle, M., et al. (2013). The cancer imaging archive (tcia): maintaining and operating a public information repository. *Journal of digital imaging*, 26:1045–1057.
- [48] Conejero, J. et al. (2018). Task-based programming in compss to converge from hpc to big data. *IJHPCA*, 32(1):45–60.
- [49] Corrales, M., Fenwick, M., and Forgó, N. (2017). *New technology, big data and the law*. Springer.
- [50] Cosmina, I. (2017). Spring microservices with spring cloud. In *Pivotal Certified Professional Spring Developer Exam*, pages 435–459. Springer.
- [51] Couper, M. P. (2011). The future of modes of data collection. *Public Opinion Quarterly*, 75(5):889–908.
- [52] Daemen, J. and Rijmen, V. (2001). Reijndael: The advanced encryption standard. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 26(3):137–139.
- [53] Daki, Hannani, E., Aqqal, Haidine, and Dahbi (2017). Big data management in smart grid: concepts, requirements and implementation. *Journal of Big Data*, 4(1):1–19.
- [54] Dastjerdi, A. V. and Buyya, R. (2016). Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116.
- [55] Deelman, E. (2010). Grids and clouds: Making workflow applications work in heterogeneous distributed environments. *The International Journal of High Performance Computing Applications*, 24(3):284–298.

- [56] Deelman, E. et al. (2019). The evolution of the pegasus workflow management software. *CiSE*, 21(4):22–36.
- [57] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., da Silva, R. F., Livny, M., et al. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.
- [58] del Rio Astorga, D., Dolz, M. F., Fernández, J., and García, J. D. (2017). A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175.
- [59] Dhaouadi, A., Bousselmi, K., Gammoudi, M. M., Monnet, S., and Hammoudi, S. (2022). Data warehousing process modeling from classical approaches to new trends: Main features and comparisons. *Data*, 7(8):113.
- [60] Di Tommaso, P., Chatzou, M., Floden, E. W., Barja, P. P., Palumbo, E., and Notredame, C. (2017). Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319.
- [61] Dictionarie, O. (2022). Agnosticism. <https://www.oxfordlearnersdictionaries.com/definition/english/agnosticism?q=agnosticism>.
- [62] Ding, Z., Wang, S., and Pan, M. (2020). Qos-constrained service selection for networked microservices. *IEEE Access*, 8:39285–39299.
- [63] Domingo-Ferrer, J., Farras, O., Ribes-González, J., and Sánchez, D. (2019). Privacy-preserving cloud computing on sensitive data: A survey of methods, products and challenges. *Computer Communications*, 140:38–60.
- [64] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and

- Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.
- [65] Dubey, A. and Wagle, D. (2007). Delivering software as a service. *The McKinsey Quarterly*, 6(2007):2007.
- [66] ECOSUR (July 28, 2020). Eris. <https://www.ecosur.mx/tag/antena-eris/>.
- [67] El-Sappagh, S. H. A., Hendawi, A. M. A., and El Bastawissy, A. H. (2011). A proposed model for data warehouse etl processes. *Journal of King Saud University-Computer and Information Sciences*, 23(2):91–104.
- [68] Elliott, D., Otero, C., Ridley, M., and Merino, X. (2018). A cloud-agnostic container orchestrator for improving interoperability. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 958–961. IEEE.
- [69] Fajardo, E., Dost, J., Holzman, B., Tannenbaum, T., Letts, J., Tiradani, A., Bockelman, B., Frey, J., and Mason, D. (2015). How much higher can htcondor fly? In *JPCS*, volume 664, page 062014.
- [70] Fowler, J. W. and Mönch, L. (2022). A survey of scheduling with parallel batch (p-batch) processing. *European journal of operational research*, 298(1):1–24.
- [71] Framework, D. N. B. D. I. (2015). Draft nist big data interoperability framework: Volume 1, definitions. *NIST Special Publication*, 1500:1.
- [72] Freitas, A. and Curry, E. (2016). Big data curation. *New horizons for a data-driven economy: A roadmap for usage and exploitation of big data in Europe*, pages 87–118.
- [73] Fry, R., Berry, R., Higgs, G., Orford, S., and Jones, S. (2012). The wiserd geoportal: A tool for the discovery, analysis and visualization of socio-economic (meta-) data for wales. *Transactions in GIS*, 16(2):105–124.

- [74] Gai, K. (2014). A review of leveraging private cloud computing in financial service institutions: Value propositions and current performances. *Int. J. Comput. Appl.*, 95(3):40–44.
- [75] Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.
- [76] Gao, M., Chen, M., Liu, A., Ip, W. H., and Yung, K. L. (2020). Optimization of microservice composition based on artificial immune algorithm considering fuzziness and user preference. *IEEE Access*, 8:26385–26404.
- [77] Gao, Z., Cecati, C., and Ding, S. X. (2015). A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches. *IEEE transactions on industrial electronics*, 62(6):3757–3767.
- [78] Garg, N. (2013). *Apache kafka*. Packt Publishing Birmingham, UK.
- [79] Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448.
- [80] Goecks, J., Nekrutenko, A., and Taylor, J. (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86.
- [81] Goel, N., Yadav, A., and Singh, B. M. (2016). Medical image processing: a review. *2016 Second International Innovative Applications of Computational Intelligence on Power, Energy and Controls with their Impact on Humanity (CIPECH)*, pages 57–62.
- [82] Gonzalez, J., Perez, J., Sosa, V., Sanchez, L., and Bergua, B. (2015). Skycds: A resilient content delivery service based on diversified cloud storage. *SIMPAT*, 54:64–85.

- [83] Gonzalez-Compean, J., Sosa-Sosa, V., Diaz-Perez, A., Carretero, J., and Yanez-Sierra, J. (2018). Sacbe: A building block approach for constructing efficient and flexible end-to-end cloud storage. *Journal of Systems and Software*, 135:143–156.
- [84] Google (2019). Istio. <https://cloud.google.com/istio/?hl=es>.
- [85] Google (2023). Workflows. <https://cloud.google.com/workflows>.
- [86] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). Spar: a dsl for high-level and productive stream parallelism. *Parallel Processing Letters*, 27(01):1740005.
- [87] Haile, N. and Altmann, J. (2018). Evaluating investments in portability and interoperability between software service platforms. *FIGS*, 78:224–241.
- [88] Harzenetter, L., Breitenbücher, U., Leymann, F., Saatkamp, K., Weder, B., and Wurster, M. (2019). Automated generation of management workflows for applications based on deployment models. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 216–225. IEEE.
- [89] Hassan, H. B., Barakat, S. A., and Sarhan, Q. I. (2021). Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29.
- [90] Hausenblas, M., Bijnens, N., and Marz, N. (2015). Lambda architecture. URL: <http://lambda-architecture.net/>. *Luettu*, 6:2014.
- [91] Hecke, T. V. (2012). Power study of anova versus kruskal-wallis test. *Journal of Statistics and Management Systems*, 15(2-3):241–247.
- [92] Holm et al. (2016). Cloudflow-an infrastructure for engineering workflows in the cloud. In *UBICOMM 2016*, volume 10, pages 158–165.

- [93] Hong, L., Luo, M., Wang, R., Lu, P., Lu, W., and Lu, L. (2018). Big data in health care: Applications and challenges. *Data and information management*, 2(3):175–197.
- [94] Hosny, A., Vera-Licona, P., Laubenbacher, R., and Favre, T. (2016). Algorun: a docker-based packaging system for platform-agnostic implemented algorithms. *Bioinformatics*, 32(15):2396–2398.
- [95] Hosseini, S. M. and Arani, M. G. (2015). Fault-tolerance techniques in cloud storage: a survey. *International Journal of Database Theory and Application*, 8(4):183–190.
- [96] Hu, C., Zhu, J., Yang, R., Peng, H., Wo, T., Xue, S., Yu, X., Xu, J., and Ranjan, R. (2020). Toposch: Latency-aware scheduling based on critical path analysis on shared yarn clusters. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 619–627. IEEE.
- [97] Hui, K. L. and Chau, P. Y. K. (2002). Classifying digital products. *Commun. ACM*, 45:73–79.
- [98] Ibrahim, S., He, B., and Jin, H. (2011). Towards pay-as-you-consume cloud computing. In *2011 IEEE International Conference on Services Computing*, pages 370–377. IEEE.
- [99] IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.
- [100] Iserte, S., Prades, J., Reaño, C., and Silla, F. (2016). Increasing the performance of data centers by combining remote gpu virtualization with slurm. In *CCGrid*, pages 98–101. IEEE.
- [101] Jamshidi, P., Ahmad, A., and Pahl, C. (2013). Cloud migration research: a systematic review. *IEEE transactions on cloud computing*, 1(2):142–157.
- [102] Jiang, F., Ferriter, K., and Castillo, C. (2020). A cloud-agnostic framework to enable cost-aware scheduling of applications in a multi-cloud environment. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE.

- [103] Jindal, A., Podolskiy, V., and Gerndt, M. (2019). Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32.
- [104] John, V. and Liu, X. (2017). A survey of distributed message broker queues. *arXiv preprint arXiv:1704.00411*.
- [105] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.
- [106] Joseph, C. T. and Chandrasekaran, K. (2020). Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments. *Journal of Systems Architecture*, 111:101785.
- [107] Karim, M. R. and Alla, S. (2017). *Scala and Spark for Big Data Analytics: Explore the concepts of functional programming, data streaming, and machine learning*. Packt Publishing Ltd.
- [108] Kasztelnik, M. et al. (2017). Support for taverna workflows in the vph-share cloud platform. *Computer Methods and Programs in Biomedicine*, 146:37–46.
- [109] Kim, G.-H., Trimi, S., and Chung, J.-H. (2014). Big-data applications in the government sector. *Communications of the ACM*, 57(3):78–85.
- [110] Koo, J., Faseeh Qureshi, N. M., Siddiqui, I. F., Abbas, A., and Bashir, A. K. (2020). IoT-enabled directed acyclic graph in spark cluster. *Journal of Cloud Computing*, 9(1):1–15.
- [111] Kranjc, J., Orač, R., Podpečan, V., Lavrač, N., and Robnik-Šikonja, M. (2017). Clowdfloows: Online workflows for distributed big data mining. *FIGS*, 68:38–58.
- [112] Kumar, S. and Rai, S. (2012). Survey on transport layer protocols: Tcp & udp. *International Journal of Computer Applications*, 46(7):20–25.

- [113] Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459.
- [114] Lai, J., Deng, R. H., and Li, Y. (2011). Fully secure ciphertext-policy hiding cp-abe. In *Information Security Practice and Experience: 7th International Conference, ISPEC 2011, Guangzhou, China, May 30–June 1, 2011. Proceedings* 7, pages 24–39. Springer.
- [115] Laskey, K. B. and Laskey, K. (2009). Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(1):101–105.
- [116] Lawton, G. (2008). Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15.
- [117] Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., and Moon, B. (2012). Parallel data processing with mapreduce: a survey. *AcM SIGMOD record*, 40(4):11–20.
- [118] Li, Z., Ge, J., Yang, H., Huang, L., Hu, H., Hu, H., and Luo, B. (2016). A security and cost aware scheduling algorithm for heterogeneous tasks of scientific workflow in clouds. *Future Generation Computer Systems*, 65:140–152.
- [119] Lin, C. and Khazaei, H. (2020). Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632.
- [120] Liu, J., Pacitti, E., Valduriez, P., and Mattoso, M. (2015). A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13:457–493.
- [121] Liu, X., Iftikhar, N., and Xie, X. (2014). Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 356–361.
- [122] Lordan, Tejedor, Ejarque, Rafanell, Álvarez, Marozzo, Lezzi, Sirvent, et al. (2014). Servicess: An interoperable programming framework for the cloud. *J. Grid Comput.*, 12(1):67–91.

- [123] Lovas, R., Dózsa, G., Kacsuk, P., Podhorszki, N., and Drótos, D. (2004). Workflow support for complex grid applications: Integrated and portal solutions. In *Grid Computing*, pages 129–138. Springer.
- [124] Luksa, M. (2017). *Kubernetes in action*. Simon and Schuster.
- [125] Madduri et al. (2015). The globus galaxies platform: delivering science gateways as a service. *CCPE*, 27(16):4344–4360.
- [126] Maier, M. W., Emery, D., and Hilliard, R. (2001). Software architecture: Introducing ieee standard 1471. *Computer*, 34(4):107–109.
- [127] Mallawaarachchi, V. (2018). 10 common software architectural patterns in a nutshell.
- [128] Manivannan, M., Negi, A., and Stenström, P. (2013). Efficient forwarding of producer-consumer data in task-based programs. In *2013 42nd International Conference on Parallel Processing*, pages 517–522. IEEE.
- [129] Mansoori, B., Erhard, K. K., and Sunshine, J. L. (2012). Picture archiving and communication system (pacs) implementation, integration & benefits in an integrated health system. *Academic radiology*, 19(2):229–235.
- [130] Manuel, P. D. and AlGhamdi, J. (2003). A data-centric design for n-tier architecture. *Information Sciences*, 150(3-4):195–206.
- [131] Mattoso, M., Dias, J., Ocana, K. A., Ogasawara, E., Costa, F., Horta, F., Silva, V., and De Oliveira, D. (2015). Dynamic steering of hpc scientific workflows: A survey. *Future Generation Computer Systems*, 46:100–113.
- [132] Maximilien, E. M., Ranabahu, A., Engehausen, R., and Anderson, L. C. (2009). Toward cloud-agnostic middlewares. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 619–626.

- [133] McKight, P. E. and Najab, J. (2010). Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1.
- [134] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing. *National Institute of Standards and Technology*.
- [135] Metaflow (2022). Metaflow. <https://docs.metaflow.org/>.
- [136] Mohan and Kangasharju (2016). Edge-fog cloud: a distributed cloud for internet of things computations. In *8th CloT*, pages 1–6. IEEE.
- [137] MoleculerJS (2022). Moleculer, progressive microservices framework for node.js. <https://moleculer.services/>.
- [138] Montella, R., Di Luccio, D., Ciaramella, A., and Foster, I. (2019). Stormseeker: A machine-learning-based mediterranean storm tracer. In *IDCS 2019*, pages 444–456. Springer.
- [139] Montella, R., Di Luccio, D., and Kosta, S. (2018). Dagon*: Executing direct acyclic graphs as parallel jobs on anything. In *2018 WORKS*, pages 64–73. IEEE.
- [140] Morales et al. (2018). A data distribution service for cloud and containerized storage based on information dispersal. In *SOSE*, pages 86–95. IEEE.
- [141] Morales-Sandoval, M., Gonzalez-Compean, J. L., Diaz-Perez, A., and Sosa-Sosa, V. J. (2018). A pairing-based cryptographic approach for data security in the cloud. *International Journal of Information Security*, 17(4):441–461.
- [142] Munappy, A. R., Bosch, J., and Olsson, H. H. (2020). Data pipeline management in practice: Challenges and opportunities. In *Product-Focused Software Process Improvement: 21st International Conference, PROFES 2020, Turin, Italy, November 25–27, 2020, Proceedings 21*, pages 168–184. Springer.

- [143] Naas, M. I., Parvedy, P. R., Boukhobza, J., and Lemarchand, L. (2017). ifogstor: an iot data placement strategy for fog infrastructure. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 97–104. IEEE.
- [144] Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.
- [145] Nguyen, P. and Nahrstedt, K. (2017). Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows. In *2017 IEEE ICAC*, pages 187–196. IEEE.
- [146] Nupponen, J. and Taibi, D. (2020). Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50. IEEE.
- [147] Nwokeji, J. C., Aqlan, F., Anugu, A., and Olagunju, A. (2018). Big data etl implementation approaches: A systematic literature review (p). In *SEKE*, pages 714–713.
- [148] O'Leary, D. E. (2013). Artificial intelligence and big data. *IEEE intelligent systems*, 28(2):96–99.
- [149] Ongaro, D. and Ousterhout, J. (2015). The raft consensus algorithm. *Lecture Notes CS*, 190:2022.
- [150] Onken, M., Eichelberg, M., Riesmeier, J., and Jensch, P. (2010). Digital imaging and communications in medicine. In *Biomedical Image Processing*, pages 427–454. Springer.
- [151] Opara, J., Sahandi, R., and Tian, F. (2014). Critical review of vendor lock-in and its impact on adoption of cloud computing. In *i-Society 2014*, pages 92–97. IEEE.
- [152] Opara-Martins, J. (2018). Taxonomy of cloud lock-in challenges. In *Mobile Computing-Technology and Applications*. IntechOpen.

- [153] Opara-Martins, J. et al. (2016). Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, 5(1):1–18.
- [154] Oracle (2022). Helidon project. <https://helidon.io/>.
- [155] Ortega, J. (2010). *Patterns for Parallel Software Design*. Wiley Publishing, 1st edition.
- [156] Pang, B., Nijkamp, E., and Wu, Y. N. (2020). Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248.
- [157] Panja, A., Christy, J. J., and Abdul, Q. M. (2021). An approach to skin cancer detection using keras and tensorflow. In *Journal of Physics: Conference Series*, volume 1911, page 012032. IOP Publishing.
- [158] Papageorgiou, A., Cheng, B., and Kovacs, E. (2015). Real-time data reduction at the network edge of internet-of-things systems. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 284–291. IEEE.
- [159] Paschali, M. et al. (2017). Reusability of open source software across domains: A case study. *JSS*, 134:211–227.
- [160] Pathan, A.-M. K., Buyya, R., et al. (2007). A taxonomy and survey of content delivery networks. *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, 4(2007):70.
- [161] Pautasso, C. and Alonso, G. (2006). Parallel computing patterns for grid workflows. In *2006 Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE.
- [162] Platnick, S., Meyer, K. G., King, M. D., Wind, G., Amarasinghe, N., Marchant, B., Arnold, G. T., Zhang, Z., Hubanks, P. A., Holz, R. E., et al. (2016). The modis cloud optical and microphysical products: Collection 6 updates and examples from terra and aqua. *IEEE Transactions on Geoscience and CiSEsing*, 55(1):502–525.

- [163] Ponge, J. (2020). *Vert. x in Action*. Manning Publications.
- [164] Qasha, R., Cała, J., and Watson, P. (2016). A framework for scientific workflow reproducibility in the cloud. In *2016 IEEE 12th International Conference on e-Science (e-Science)*, pages 81–90. IEEE.
- [165] Qiu, J., Wu, Q., Ding, G., Xu, Y., and Feng, S. (2016). A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016:1–16.
- [166] Quoc, D. L., Chen, R., Bhatotia, P., Fetze, C., Hilt, V., and Strufe, T. (2017). Approximate stream analytics in apache flink and apache spark streaming. *arXiv preprint arXiv:1709.02946*.
- [167] Rabin, M. O. (1990). The information dispersal algorithm and its applications. In *Sequences: Combinatorics, Compression, Security, and Transmission*, pages 406–419. Springer.
- [168] Rai, R., Sahoo, G., and Mehfuz, S. (2015). Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration. *SpringerPlus*, 4:1–12.
- [169] Rawat, D. B. et al. (2019). Cybersecurity in big data era: From securing big data to data-driven security. *IEEE Transactions on Services Computing*.
- [170] Reinsel, Gantz, and Rydning (2018). The digitization of the world: from edge to core. *Framingham: International Data Corporation*.
- [171] Reyes, Gonzalez, Sosa, Carretero, and Garcia (2020). Kulla, a container-centric construction model for building infrastructure-agnostic distributed and parallel applications. *JSS*, page 110665.
- [172] Richards, M. (2015a). *Microservices vs. service-oriented architecture*. O'Reilly Media.
- [173] Richards, M. (2015b). *Software Architecture Patterns*. O'Reilly Media, Inc.
- [174] Rodriguez, M. and Buyya, R. (2017). A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments. *CCPE*, 29(8):e4041.

- [175] Sadiku, M., Shadare, A. E., Musa, S. M., Akujuobi, C. M., and Perry, R. (2016). Data visualization. *International Journal of Engineering Research And Advanced Technology (IJERAT)*, 2(12):11–16.
- [176] Safaei, A. A. (2017). Real-time processing of streaming big data. *Real-Time Systems*, 53:1–44.
- [177] Şahin, S. and Gedik, B. (2018). C-stream: a co-routine-based elastic stream processing engine. *ACM Transactions on Parallel Computing (TOPC)*, 4(3):1–27.
- [178] Salloum, S., Dautov, R., Chen, X., Peng, P. X., and Huang, J. Z. (2016). Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3):145–164.
- [179] Sánchez, D. et al. (2020). On the continuous processing of health data in edge-fog-cloud computing by using micro/nanoservice composition. *IEEE Access*, 8:120255–120281.
- [180] Sánchez-Gallegos, D. D. et al. (2021). An efficient pattern-based approach for workflow supporting large-scale science: The dagonstar experience. *FGCS*, 122:187–203.
- [181] Santiago, M., Gonzalez, J., et al. (2020). A gearbox model for processing large volumes of data by using pipeline systems encapsulated into virtual containers. *FIGS*, 106:304–319.
- [182] Serbanescu, M., Placinta, V., Hutanu, O., and Ravariu, C. (2017). Smart, low power, wearable multi-sensor data acquisition system for environmental monitoring. In *2017 10th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, pages 118–123. IEEE.
- [183] Serrano, N., Gallardo, G., and Hernantes, J. (2015). Infrastructure as a service and cloud technologies. *IEEE Software*, 32(2):30–36.
- [184] Shahrad, M., Balkind, J., and Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 1063–1075.

- [185] Shailaja, K., Seetharamulu, B., and Jabbar, M. (2018). Machine learning in healthcare: A review. In *2018 Second international conference on electronics, communication and aerospace technology (ICECA)*, pages 910–914. IEEE.
- [186] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646.
- [187] Shuaib, M., Samad, A., Alam, S., and Siddiqui, S. T. (2019). Why adopting cloud is still a challenge?—a review on issues and challenges for cloud migration in organizations. *Ambient Communications and Computer Systems: RACCCS-2018*, pages 387–399.
- [188] Siddiqa, A., Karim, A., and Gani, A. (2017). Big data storage technologies: a survey. *Frontiers of Information Technology & Electronic Engineering*, 18:1040–1070.
- [189] Sikeridis, D., Papapanagiotou, I., Rimal, B. P., and Devetsikiotis, M. (2017). A comparative taxonomy and survey of public cloud infrastructure vendors. *arXiv preprint arXiv:1710.01476*.
- [190] Şimşit, Z. T., Günay, N. S., and Vayvay, Ö. (2014). Theory of constraints: A literature review. *Procedia-Social and Behavioral Sciences*, 150:930–936.
- [191] Singh, C., Gaba, N. S., Kaur, M., and Kaur, B. (2019). Comparison of different ci/cd tools integrated with cloud platform. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 7–12. IEEE.
- [192] Smart, J. F. (2011). *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc.
- [193] Soppelsa, F. and Kaewkasi, C. (2016). *Native docker clustering with swarm*. Packt Publishing Ltd.
- [194] Staff, C. b. V. S., Naparstek, D., Burke, M., and Papazov, Y. (2022). What does cloud agnostic mean? <https://blogs.vmware.com/cloudhealth/what-does-cloud-agnostic-mean/>.

- [195] Štefanič, P., Cigale, M., Jones, A. C., Knight, L., Taylor, I., Istrate, C., Suciu, G., Ulisses, A., Stankovski, V., Taherizadeh, S., et al. (2019). Switch workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications. *Future Generation Computer Systems*, 99:197–212.
- [196] Stergiou, C., Psannis, K., Gupta, B., and Ishibashi, Y. (2018). Security, privacy & efficiency of sustainable cloud computing for big data & iot. *SUSCOM*, 19:174–184.
- [197] Stojmenovic, I. and Wen, S. (2014). The fog computing paradigm: Scenarios and security issues. In *2014 federated conference on computer science and information systems*, pages 1–8. IEEE.
- [198] Sánchez-Gallegos, D. D., Gonzalez-Compean, J. L., Alvarado-Barrientos, S., Sosa-Sosa, V. J., Tuxpan-Vargas, J., and Carretero, J. (2018). A containerized service for clustering and categorization of weather records in the cloud. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 26–31.
- [199] Taher, N. C., Mallat, I., Agoulmine, N., and El-Mawass, N. (2019). An iot-cloud based solution for real-time and batch processing of big data: Application in healthcare. In *2019 3rd international conference on bio-engineering for smart technologies (BioSMART)*, pages 1–8. IEEE.
- [200] Tang, B. and Fedak, G. (2012). Analysis of data reliability tradeoffs in hybrid distributed storage systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1546–1555. IEEE.
- [201] Taufer et al. (2017). Scheduling dag-based workflows on single cloud instances: High-performance and cost effectiveness with a static scheduler. *The International Journal of High Performance Computing Applications*, 31(1):19–31.

- [202] Taylor, I., Shields, M., Wang, I., and Harrison, A. (2007). The Triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer.
- [203] Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R. M., Torres, J., Cortes, T., and Labarta, J. (2017). Pycompss: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications*, 31(1):66–82.
- [204] Thierer, A. and Castillo, A. (2015). Projecting the growth and economic impact of the internet of things. *George Mason University, Mercatus Center, June*, 15.
- [205] Truica, C.-O., Radulescu, F., Boicea, A., and Bucur, I. (2015). Performance evaluation for crud operations in asynchronously replicated document oriented database. In *2015 20th International Conference on Control Systems and Computer Science*, pages 191–196. IEEE.
- [206] Tsai, Y.-L., Huang, K.-C., Chang, H.-Y., Ko, J., Wang, E. T., and Hsu, C.-H. (2012). Scheduling multiple scientific and engineering workflows through task clustering and best-fit allocation. In *Services 2012*, pages 1–8. IEEE.
- [207] Tychalas, D. and Karatza, H. (2021). Samw: a probabilistic meta-heuristic algorithm for job scheduling in heterogeneous distributed systems powered by microservices. *Cluster Computing*, pages 1–25.
- [208] USGS (July 31, 2020). Earthexplorer - usgs. <https://earthexplorer.usgs.gov/>.
- [209] Van der Aalst, W. and Damiani, E. (2015). Processes meet big data: Connecting data science with process science. *IEEE Transactions on Services Computing*, 8(6):810–819.
- [210] van der Zeeuw, A., van Deursen, A. J., and Jansen, G. (2022). The orchestrated digital inequalities of the iot: How vendor lock-in hinders and playfulness creates iot benefits in every life. *new media & society*, page 14614448221138075.

- [211] Varia, J., Mathew, S., et al. (2014). Overview of amazon web services. *Amazon Web Services*, 105.
- [212] Vermote, E. et al. (2016). Preliminary analysis of the performance of the landsat 8/oli land surface reflectance product. *RSE*, 185:46–56.
- [213] Viceconti, M., Hunter, P., and Hose, R. (2015). Big data, big knowledge: big data for personalized healthcare. *IEEE journal of biomedical and health informatics*, 19(4):1209–1215.
- [214] Vogel, A., Mencagli, G., Griebler, D., Danelutto, M., and Fernandes, L. G. (2021). Towards on-the-fly self-adaptation of stream parallel patterns. In *PDP*, pages 89–93. IEEE.
- [215] Walls, C. (2015). *Spring Boot in action*. Simon and Schuster.
- [216] Wang, B., Wang, C., Song, Y., Cao, J., Cui, X., and Zhang, L. (2020a). A survey and taxonomy on workload scheduling and resource provisioning in hybrid clouds. *Cluster Computing*, 23:2809–2834.
- [217] Wang, S., Ding, Z., and Jiang, C. (2020b). Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):98–115.
- [218] Wang, T., Ma, H., Zhou, Y., Zhang, R., and Song, Z. (2019). Fully accountable data sharing for pay-as-you-go cloud scenes. *IEEE Transactions on Dependable and Secure Computing*, 18(4):2005–2016.
- [219] Wang, W. and Casale, G. (2014). Evaluating weighted round robin load balancing for cloud web services. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 393–400. IEEE.
- [220] Williamson, B., Gulson, K. N., Perrotta, C., and Witzenberger, K. (2022). Amazon and the new global connective architectures of education governance. *Harvard Educational Review*, 92(2):231–256.

- [221] Wozniak, J. M., Armstrong, T. G., Wilde, M., Katz, D. S., Lusk, E., and Foster, I. T. (2013). Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *CCGrid*, pages 95–102. IEEE.
- [222] Wulder, M. A., Loveland, T. R., Roy, D. P., Crawford, C. J., Masek, J. G., Woodcock, C. E., Allen, R. G., Anderson, M. C., Belward, A. S., Cohen, W. B., et al. (2019). Current status of landsat program, science, and applications. *RSE*, 225:127–147.
- [223] Yi, S., Li, C., and Li, Q. (2015). A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42.
- [224] Yu, T., Wang, X., and Shami, A. (2017). A novel fog computing enabled temporal data reduction scheme in iot systems. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–5. IEEE.
- [225] Zampetti, F., Geremia, S., Bavota, G., and Di Penta, M. (2021). Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE.
- [226] Zheng, C. and Thain, D. (2015). Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *HPDC'15*, pages 31–38.
- [227] Zhou, L., Fu, A., Yu, S., Su, M., and Kuang, B. (2018). Data integrity verification of the outsourced big data in the cloud environment: A survey. *Journal of Network and Computer Applications*, 122:1–15.
- [228] Zwattendorfer, B. and Tauber, A. (2013). The public cloud for e-government. *International Journal of Distributed Systems and Technologies (IJDST)*, 4(4):1–14.