Implementación de una Infraestructura Escalable Basada en Contenedores y Procesamiento de Datos Distribuido para un Comercio Electrónico

Michel Dahiana Burgos Santos, Juan David Daza Rivera, Liseth Esmeralda Erazo Varela y Natalia López Gallego.

[[1]](#footnote-1)

***Abstract*—** **Este documento describe el desarrollo de una infraestructura escalable para el despliegue de una aplicación web basada en microservicios y el procesamiento de grandes volúmenes de datos asociados a la misma. La aplicación web es un comercio electrónico para la cadena minorista “MiPunto”, la cual contiene un panel de administración para gestionar productos y facturas, y una interfaz para que los clientes naveguen por el catálogo, gestionen su carrito y generen facturas. Esta aplicación fue empaquetada y desplegada en un clúster de contenedores para garantizar su escalabilidad y balanceo de carga, permitiendo el manejo de múltiples solicitudes de usuarios simultáneamente. Además, se diseñó y desplegó un clúster de procesamiento de datos distribuidos para analizar datos asociados a las ventas del comercio electrónico y generar reportes detallados en un dashboard accesible desde el panel de administración. La metodología empleada incluyó la selección de un dataset para simular el análisis de datos, la evaluación de alternativas tecnológicas, la definición de la arquitectura necesaria, y la** **ejecución de pruebas sobre la solución implementada.**

***Index Terms*— API REST, arquitectura, aplicación web, balanceo de carga, comercio electrónico, clúster de contenedores, microservicios, procesamiento de datos distribuido.**

# I. INTRODUCCIÓN

M

iPunto, una cadena minorista emergente en Colombia, ha tenido éxito en el mercado físico, pero enfrentaba desafíos en el ámbito digital debido a la falta de una plataforma robusta de comercio electrónico. Para abordar esto, se desarrolló una aplicación web basada en microservicios, que permite a los clientes navegar por el catálogo, gestionar su carrito de compras y realizar pedidos en línea, mientras que los administradores pueden gestionar el inventario y revisar facturas.

Sin embargo, ahora se enfrentan dos nuevos desafíos: por un lado, la gestión de múltiples solicitudes de usuarios simultáneamente; por el otro,el procesamiento de datos a gran escala (Big Data) de la aplicación para obtener insights valiosos sobre las ventas de la tienda. Pues a medida que aumenta el uso de recursos, la carga en los servidores también crece exponencialmente, haciendo del balanceo de carga una necesidad fundamental; además se debe tener en cuenta que las aplicaciones que manejan grandes volúmenes de datos requieren de una gran cantidad de recursos y entornos para gestionar, procesar y analizarlos de manera distribuida. En este sentido, la contenedorización es una solución que facilita el cumplimiento de estos requisitos, permitiendo ajustar los componentes de la aplicación empaquetados en contenedores de manera precisa y rápida según la demanda de los servicios [1].

Con esto en mente, el objetivo para una nueva fase del proyecto es implementar una infraestructura que soporte el despliegue y empaquetamiento de la aplicación inicial basada en microservicios y que permita además el procesamiento de datos a gran escala, de manera distribuida. Esta iniciativa pretende mejorar la funcionalidad y el rendimiento de la aplicación «MiPunto», garantizando que satisfaga las demandas tanto de los administradores como de los clientes. El alcance de este proyecto abarca la selección de un dataset para simular el análisis de datos, el diseño de la arquitectura necesaria y su implementación para cumplir el objetivo propuesto, además de la ejecución de pruebas de escalabilidad y desempeño sobre la solución implementada.

# II. ANÁLISIS

## A. Selección del dataset objetivo

El dataset seleccionado para brindar información relevante en el desarrollo de la solución se titula “Retail Sales and Customer Behavior Analysis”. Este se compone de 10.000 filas y 78 columnas que simulan el entorno de ventas de un distribuidor minorista con cuatro puntos de venta. Los datos detallan información sobre usuarios, transacciones, y productos, además de métricas sobre el comportamiento de los consumidores [2]. Dado el problema identificado y su solución desarrollada inicialmente, se decidió utilizar el dataset como base para ajustarlo a las necesidades del modelo de datos de la aplicación [12] por medio de Python. Se conservaron 8 de las columnas originales (customer\_id, product\_id, product\_category, quantity, unit\_price, product\_name, product\_stock, y customer\_city); se separó la información de los productos y de los usuarios en dos datasets, y se añadió o modificó información a cada uno de estos para complementarlos. Pues, el dataset original no contenía nombres específicos de productos o ciudades (por ejemplo, se encontraba el “Producto B”), y había campos que resultaban necesarios para el funcionamiento a nivel de la lógica del negocio y de datos, como el “username” y la “contraseña” de los usuarios. A partir de los dos datasets obtenidos anteriormente se generaron 2 datasets relacionados de facturas e ítems contenidos en las facturas (productos), sobre los cuales se fundamenta el análisis dispuesto en la segunda parte del proyecto. El dataset de facturas contiene información sobre ventas efectuadas, con columnas com id\_factura, user\_id, email, nombre, ciudad, direccion, docimento\_identidad, subtotal, precio\_envio, total y fecha; tiene 5000 registros. Por su parte el dataset de ítems contenidos en las facturas, contiene información sobre los productos asociados a las facturas con columnas como id\_item\_factura, factura\_id, product\_id, price y quantity; tiene 15.000 registros.

## B. Generación y selección de alternativas de solución

*1) Orquestación*

Se evaluaron las distintas tecnologías de orquestación como Kubernetes, Docker Swarm, AWS ECS y Apache Mesos. La selección consideró las características técnicas, la escalabilidad y la facilidad de integración con el ecosistema de trabajo, así como la capacidad de administrar una infraestructura de contenedores que garantiza un despliegue estable. En la siguiente tabla se evidencia una comparativa de las alternativas evaluadas, a partir de lo referenciado en [4],[5], [6] y [7].

TABLA I

Comparación de las tecnologías de orquestación

|  |  |  |
| --- | --- | --- |
| **Tecnología** | **Ventajas** | **Desventajas** |
| **Kubernetes** | Amplia escalabilidad, gran comunidad de soporte, automatización avanzada, permite el despliegue de aplicaciones complejas. | Curva de aprendizaje alta; configuración y mantenimiento complejos; requiere más recursos y esfuerzo de implementación. |
| **Docker Swarm** | Integración nativa con Docker, configuración sencilla, ideal para despliegues rápidos de aplicaciones; menor complejidad en clusters pequeños y medianos. | Menor capacidad de escalabilidad y características avanzadas en comparación con Kubernetes; limitado para grandes sistemas. |
| **AWS ECS** | Servicio gestionado que reduce la administración, integración nativa con otros servicios AWS, ideal para usuarios de Amazon Cloud. | Dependencia de AWS; menos control sobre la infraestructura; costos adicionales en comparación con soluciones de código abierto. |
| **Apache Mesos** | Flexibilidad para trabajar con distintos frameworks y aplicaciones distribuidas, buena gestión de recursos en clusters grandes y heterogéneos. | Complejidad de instalación; menor popularidad y comunidad en comparación con Kubernetes y Docker Swarm. |

*2) Procesamiento de datos distribuido*

 Para el procesamiento de datos distribuido se consideraron varias tecnologías como Spark, Hadoop, Apache Flink y Dask. En función de las características técnicas, adaptabilidad al entorno y la capacidad de manejo de un volumen de datos que se adapta a la cantidad de datos obtenidos. En la siguiente tabla se evidencia una comparativa de las alternativas evaluadas, a partir de lo referenciado en [8],[9], [10] y [11].

TABLA II

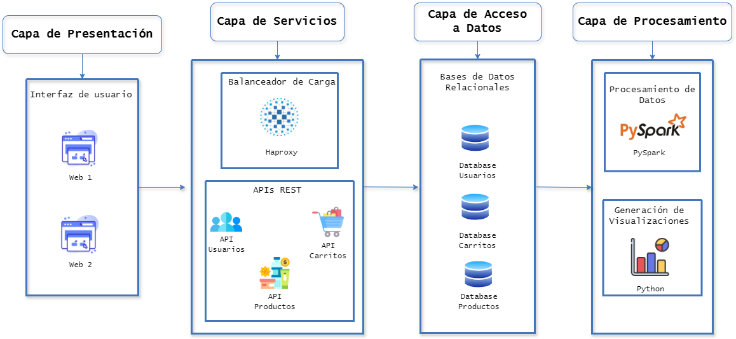
Comparación de las tecnologías de procesamiento de datos distribuido

|  |  |  |
| --- | --- | --- |
| **Tecnología** | **Ventajas** | **Desventajas** |
| **Apache Spark** | Procesamiento en memoria rápido, API intuitiva, soporte para procesamiento en tiempo real y análisis avanzado (Spark SQL) | Requiere una gestión de memoria cuidadosa en clusters grandes, lo que puede requerir ajustes adicionales. |
| **Hadoop** | Plataforma madura para procesamiento batch; HDFS eficiente para almacenar grandes volúmenes de datos; MapReduce gestiona bien tareas intensas. | Enfoque en almacenamiento en disco, lo que reduce la velocidad; mayor complejidad en implementación y administración. |
| **Apache Flink** | Ideal para procesamiento en tiempo real con baja latencia; maneja tanto flujos en tiempo real como datos batch. | Menor ecosistema y soporte en machine learning comparado con Spark; comunidad y recursos de desarrollo limitados. |
| **Dask** | Compatible con bibliotecas Python, fácil de implementar, buena escalabilidad en proyectos de mediana envergadura. | Menor rendimiento en clusters grandes; carece de módulos integrados para machine learning y procesamiento en tiempo real. |

*3) Solución seleccionada*

Debido a la facilidad de configuración y simplicidad en entornos que no requieren una arquitectura compleja, como el caso de la tienda «MiPunto», Docker Swarm fue escogido como la opción ideal para la orquestación de contenedores. La integración con Docker permite una transición fluida, favoreciendo el despliegue de los microservicios en entornos de desarrollo ágil. Así mismo, Swarm permite capacidades de escalabilidad horizontal, lo que permite una distribución de carga acorde a las necesidades de la aplicación, optimizando los recursos.  En el caso de procesamiento distribuido de datos, Apache Spark garantiza la efectividad de la aplicación debido a su alto rendimiento en el manejo de datos en memoria RAM y su ecosistema versátil que permiten el análisis de datos estructurado. Estas capacidades se alinean con los requerimientos de análisis del proyecto, ya que permiten la rápida manipulación y consulta de los datasets. Adicionalmente, Spark presenta una compatibilidad con Python, lo que se integra con las herramientas y lenguajes ya empleados en el desarrollo del proyecto, asegurando una experiencia fluida y optimizada en la gestión de datos.

## C. Definición de la arquitectura completa del sistema



**Fig. 1.** Diagrama de la arquitectura del sistema.

### 1) Componentes

La arquitectura de la aplicación se encuentra englobada en una arquitectura de tres capas, en donde hay una capa de presentación, o interfaz de usuario; una capa de lógica del negocio, donde se procesa la información; y la capa de acceso a datos, donde se almacena y gestiona la información de la aplicación [12]. En este caso las tres capas se encuentran dispuestas de la siguiente manera:

* **Capa de presentación:**dos instancias de una interfaz de *frontend.*
* **Capa de lógica del negocio:**tres microservicios de productos, usuarios y carritos de compra; y un balanceador de carga.
* **Capa de acceso a datos:** tres bases de datos de productos, usuarios y carritos de compra.

A esto se le suma una capa más, para el procesamiento y analítica de datos distribuidos asociados a la aplicación.

### 2) Servicios

La arquitectura de la aplicación se basa en microservicios, lo cual implica la división de la misma en diferentes servicios con bajo acoplamiento entre ellos y orientados a responsabilidades de negocio [13]. Además, para esta arquitectura se usa el método de base de datos privada, donde cada microservicio tiene su propia base de datos; de manera que ocurre una segmentación del almacenamiento central en un conjunto de segmentos de acuerdo al límite de negocio de cada microservicio [13]. Las funcionalidades o servicios que provee cada capa de componentes son las siguientes:

* **Capa de Presentación:** manejo de la interfaz de usuario.
* **Capa de Servicios:** gestión de la lógica de negocio para productos, usuarios y carritos de compra; y distribución de las solicitudes de los usuarios
* **Capa de Acceso a Datos:** almacenamiento y gestión de datos sobre productos, usuarios y carritos de compra.
* **Capa de Procesamiento:** procesamiento y análisis de datos distribuidos; generación de gráficas.

### 3) Tecnologías

#### 3a) Clúster de contenedores

* **Docker**: Para la creación y gestión de contenedores.
* **Docker Compose**: Para definir y ejecutar aplicaciones multi-contenedor.
* **Docker Hub**: Para almacenar y distribuir imágenes de contenedores.
* **Docker Swarm**: Para la orquestación de contenedores y su escalamiento.
* **MySQL**: Como sistema de gestión de bases de datos.
* **HAProxy**: Como balanceador de carga.
* **Apache**: Como servidor web.
* **Node.js:** Para el desarrollo de los microservicios.

#### 3b) Clúster de procesamiento de datos distribuidos

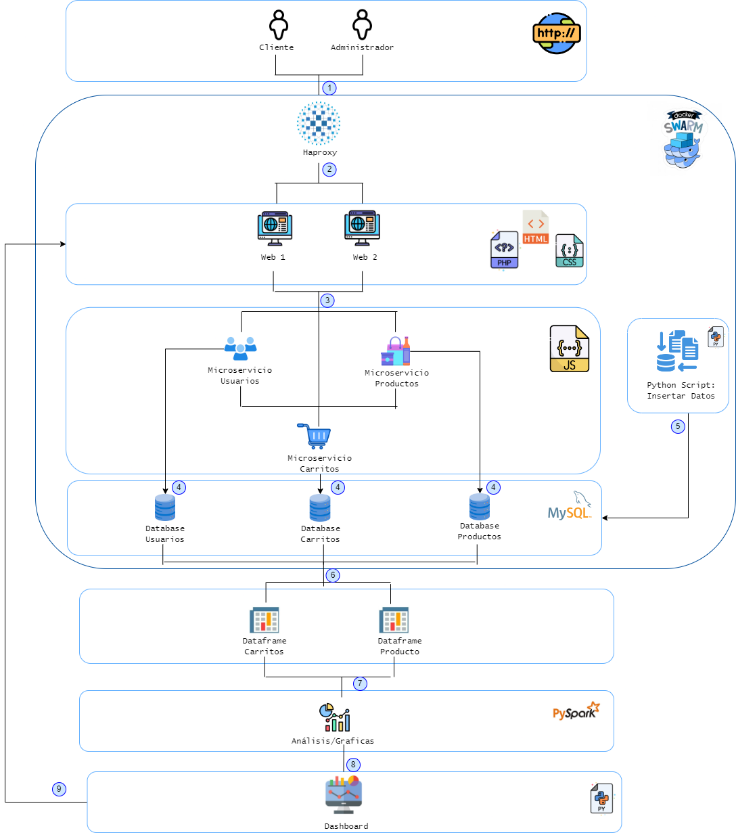
* **Apache Spark:** framework para el procesamiento de datos distribuidos.
* **Python:** lenguaje de programación para realizar el análisis de datos.
* **PySpark:** API de Python para trabajar con Apache Spark, utilizada para analizar DataFrames y generar visualizaciones y estadísticas.

# III. DISEÑO

## A. Propuesta del pipeline

### 1) Diagrama de los componentes a utilizar

El siguiente diagrama muestra los componentes y servicios que se utilizan en el proyecto, así como sus relaciones y el flujo de trabajo.



**Fig. 2.** Diagrama del pipeline del sistema.

### 3) Relación y flujo de trabajo entre los componentes

1. Los usuarios (Clientes/Administradores) Estos usuarios acceden a la aplicación enviando solicitudes HTTP (GET, POST, etc.) a través de una URL específica. Pueden realizar diversas acciones como buscar productos, ver el carrito, o gestionar productos en el caso del administrador.
2. *HAProxy* recibe las solicitudes de los usuarios y las redirige a uno de los servidores web disponibles (*Web 1* y *Web 2*), balanceándolas mediante el algoritmo *Round Robin.*
3. Los servidores web procesan las solicitudes, utilizando tecnologías como PHP, HTML y CSS, y dependiendo de la solicitud pueden requerir funcionalidades de los diferentes microservicios (Usuarios, Productos o Carritos).
4. Los Microservicios interactúan con sus respectivas bases de datos para almacenar o recuperar información (operaciones de “Crear, Leer, Actualizar y Borrar" o CRUD), y devuelven la respuesta al servidor web.
5. *Python Scripts* carga los registros de las bases de datos a partir de archivos CSV (este proceso se inicializa después de las bases de datos, cuando se despliega la aplicación).
6. Se hace uso de un conector de MysQL en *Spark* para comunicarse con las bases de datos *Database Productos* y *Database Carritos, y se* crean DataFrames a partir de estas.
7. Los DataFrames se analizan utilizando *PySpark* se corre el algoritmo *análisis.py* para generar visualizaciones y estadísticas sobre las ventas de la aplicación.
8. Las gráficas de las estadísticas son guardadas en un volumen compartido por las máquinas utilizadas en el procesamiento de distribuido de Spark y se ven reflejadas en el volumen “/var/www/html/images” de los servidores web para cargarse en la pestaña de estadísticas del usuario administrador.
9. Finalmente, al acceder a la pestaña de “estadísticas” del usuario administrador se muestra un *Dashboard* con los resultados gráficos del análisis.

### 3) Descripción de los componentes

#### 3a) Clúster de contenedores

Dado que una arquitectura basada en microservicios divide la aplicación en un conjunto de pequeños servicios independientes, cada uno de estos debe tener la capacidad de desplegarse con frecuencia y de manera autónoma. En este contexto, el empaquetamiento y despliegue de cada servicio mediante *contenedores* proporciona un mecanismo ideal para este proceso [14]. Pues, los contenedores encapsulan cada componente de una aplicación en su propio entorno con todas las dependencias necesarias; además, cada contenedor es independiente y funciona de manera consistente en cualquier entorno [15]. Esta práctica finalmente permite que la aplicación completa se despliegue como un clúster utilizando un *orquestador* [14]. En este caso los componentes de la aplicación web que están empaquetados en contenedores Docker son los siguientes:

* **HAProxy:** este contenedor sirve de balanceador de carga, actuando como punto de entrada único para las solicitudes entrantes en la aplicación y distribuyéndolas de manera equitativa entre los servicios web.
* **Web1 y Web2 (capa de presentación):** dos contenedores del *frontend* que constituyen la capa de presentación de la aplicación. Muestran información a los usuarios y recogen datos que ingresen estos para que la aplicación pueda procesarla.

TABLA III

Datos gestionados y presentados por los servicios web, diferenciados por usuario.

|  |  |  |
| --- | --- | --- |
| **Usuario** | **Tipo de información a presentar** | **Presentación** |
| **Cliente** | Catálogo de productos. | Pestaña para visualizar los productos del catálogo y añadirlos al carrito de compra. |
|  | Carrito de compras con un resumen de los productos seleccionados, cantidades y precios. También subtotal, precio de envío y total. | Pestaña para el carrito de compra que permite su gestión y la generación de una factura correspondiente. |
| **Administrador** | Gestión de productos. | Pestaña para gestionar productos. |
|  | Facturas existentes. | Pestaña para visualizar las facturas existentes. |
|  | Estadísticas. | Pestaña para visualizar imágenes resultantes del análisis de las facturas y sus ítems. |

* **Microservicios (capa de lógica del negocio)**
* **Productos:** microservicio que se encarga de gestionar las operaciones relacionadas con los productos, como agregar nuevos productos al catálogo, modificar información existente, dar de baja producto; también realizar un seguimiento del inventario y proporciona datos relevantes sobre productos al microservicio de *Carritos* cuando sea necesario. Está expuesto en el puerto "3001:3001".
* **Usuarios:** microservicio que se encarga gestionar las operaciones relacionadas con los usuarios, como validar las credenciales de acceso, otorgar y revocar permisos, almacenar información personal de los usuarios, y proporcionar datos relevantes sobre usuarios al microservicio de *Carritos* cuando sea necesario. Se expone en el puerto "3002:3002".
* **Carritos:** microservicio que se encarga de recibir la información de los productos que desean adquirir los usuarios, validar su existencia, calcular el subtotal de la compra de acuerdo a sus precios y cantidades, calcular el total con el precio de envío y crear facturas cuando se guarde el carrito para efectuar una compra, actualizando el inventario de productos. Está expuesto en el puerto "3003:3003".
  + **Python scripts:** contenedor dedicado a ejecutar scripts de Python cuyo propósito es cargar, desde archivos CSV, registros iniciales a las bases de datos.
* **Bases de Datos (capa de acceso de datos)**
  + **Database productos:** Un contenedor que aloja la base de datos relacional de MySQL para almacenar información esencial sobre los productos, como identificador único, nombre, categoría, precio unitario en pesos colombianos y la cantidad disponible en inventario. Se expone en el puerto "32000:3306".
  + **Database usuarios**: Un contenedor que aloja la base de datos relacional de MySQL para almacenar información personal básica de los usuarios, como credenciales (nombre de usuario y contraseña), correo electrónico, nombre completo, ciudad y dirección de residencia, y un documento de identidad. Se expone en el puerto "32001:3306".
  + **Database Carritos:** Un contenedor que aloja la base de datos relacional de MySQL para almacenar información en 4 tablas que almacenan sobre los carritos de compra, los itéms agregados a los carritos, las facturas y los ítems de cada factura. Se expone en el puerto "32002:3306".

Para el proceso de automatización del despliegue, la gestión, el escalado y la conexión en red de los contenedores [16], así como para su distribución en múltiples nodos, se debe utilizar un orquestador, que vendría a ser el proporcionado por Docker Swarm. En este los contenedores se programan como servicios, entendidos en este contexto como grupos escalables de contenedores con funciones de red añadidas mantenidas automáticamente por Swarm [17]. En un Swarm los nodos manager son los encargados de gestionar y coordinar todo el funcionamiento del clúster. Se ocupan de mantener un estado consistente del Swarm, tomar decisiones sobre dónde ejecutar los servicios y proporcionar una interfaz para administrar el clúster. Por otro lado, los nodos worker son los que ejecutan las tareas asignadas por los nodos manager, es decir, donde realmente se ejecutan los contenedores. Por defecto, los nodos manager también son workers [18].

Ahora bien, entendiendo que un clúster de contenedores agrupa múltiples nodos o hosts en clústeres, donde cada nodo posee varios contenedores que proporcionan servicios [14], se debe señalar que en este caso se ha configurado un solo clúster Swarm con dos nodos: un nodo manager, que se encarga de coordinar las tareas del clúster, y un nodo worker, dedicado a ejecutar las aplicaciones. En el nodo manager (IP 192.168.10.3), se han desplegado servicios esenciales como bases de datos, microservicios, scripts de Python y un balanceador de carga. Por otro lado, el nodo worker IP (192.168.10.2) se utiliza para ejecutar los servicios web, optimizando así la distribución de la carga de trabajo.

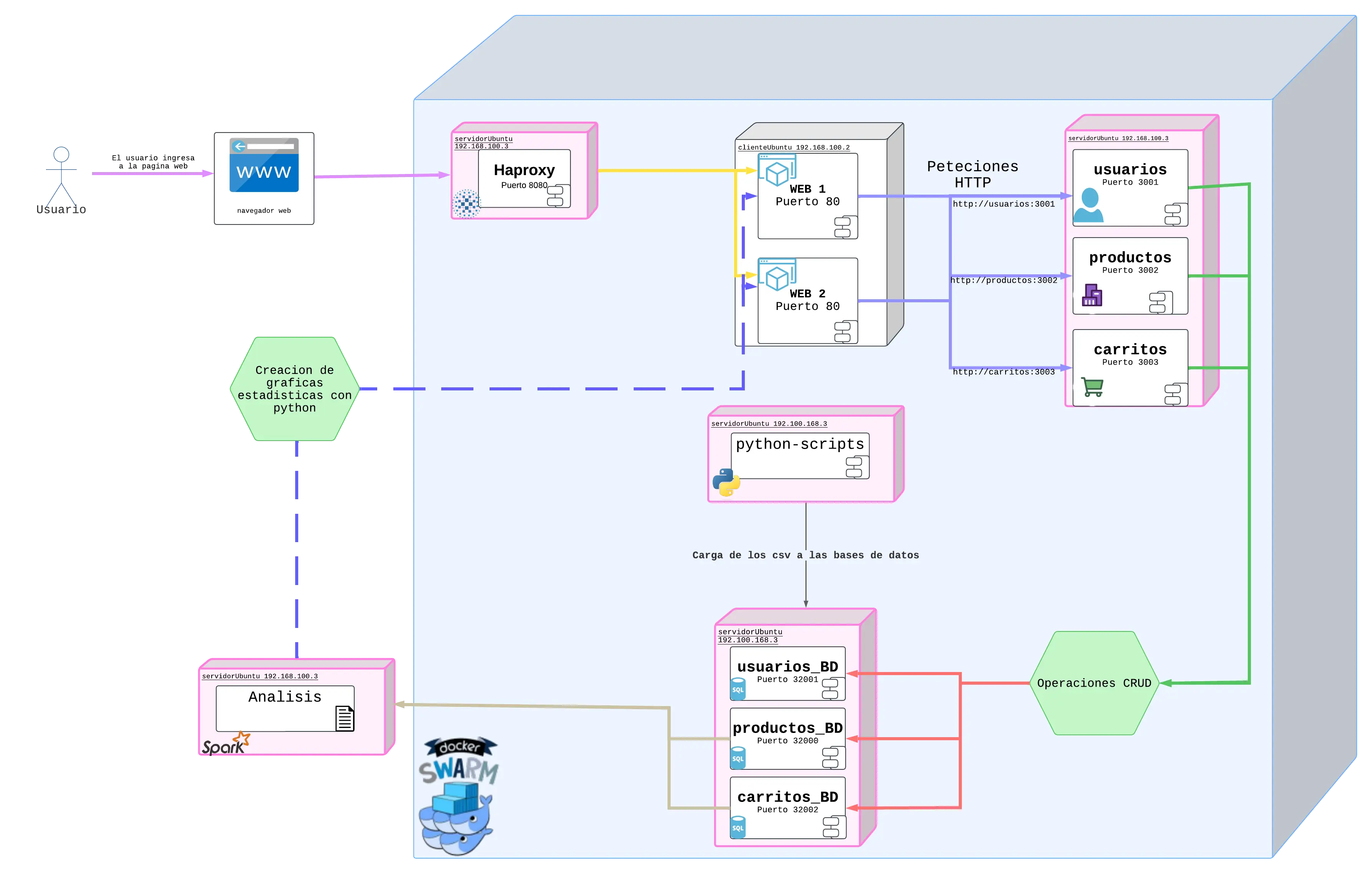
*3b) Clúster de procesamiento de datos distribuido*

La velocidad es esencial en el procesamiento de grandes volúmenes de datos, ya que puede determinar si se pueden explorar datos de manera interactiva o si se debe esperar minutos u horas. Los métodos tradicionales de análisis en este caso, como las consultas a bases de datos relacionales, a menudo no son suficientes cuando se trata de escalabilidad y rapidez [19]. Un sistema distribuido (como el provisto por Spark), o clúster, que utiliza múltiples nodos para procesar datos en paralelo permite procesar datos mucho más rápido que una sola máquina. En este sistema, un controlador central (driver), coordina los nodos (workers), distribuyendo el trabajo y gestionando la ejecución de las aplicaciones de análisis [20]. Para este proyecto se utiliza un solo clúster compuesto por dos nodos, un *Driver* (192.168.10.2) y un *Worker* (IP 192.168.10.3) y

Para efectuar el análisis primero se utiliza un *conector* de MySQL en Spark para comunicarse con las bases de datos “Database Productos” y “Database Carritos”, creando DataFrames a partir de estos datos. Estos DataFrames se analizan con *PySpark*, ejecutando el *algoritmo* “análisis.py*”* para generar visualizaciones y estadísticas sobre las ventas de la aplicación. Las gráficas resultantes, en formato en PNG y HTML, se guardan en un directorio compartido por los nodos, que actúan como *volumen* para los servidores web, y se ven reflejadas al interior de estos en la ruta “/var/www/html/images” permitiendo que se carguen en la pestaña de estadísticas del usuario administrador.

## B. Diagrama de despliegue

En el siguiente diagrama se muestra cómo se despliegan los diferentes componentes en el sistema, incluyendo los nodos (con sus direcciones IP) en los que se ejecutan, y sus puertos. Ha de señalarse que el análisis facilitado por Spark corre de manera distribuido en el nodo con la IP 192.168.10.3 y 192.168.10.2.



**Fig. 3.** Diagrama de despliegue.

# IV. IMPLEMENTACIÓN Y PRUEBAS

## A. Implementación de la solución diseñada

La aplicación desarrollada inicialmente se basa en una arquitectura de microservicios, a los cuales se accede por medio de API Rest, donde cada uno de sus componentes se empaquetan en contenedores Docker; el código fuente de estos componentes se almacena en GitHub junto a sus respectivos archivos de configuración de Docker (Dockerfiles), mientras que las imágenes de Docker generadas a partir de este código se almacenan en repositorios de Docker Hub. Para desplegar y gestionar estos contenedores de forma distribuida en un clúster (con dos nodos Master y Worker), se utiliza la herramienta de orquestación Docker Swarm, y un archivo docker-compose.yml que llama a las imágenes de los repositorios de Docker Hub. Además, se utiliza un clúster de procesamiento de datos distribuido basado en Apache Spark, con dos nodos (Driver y Worker), y su librería PySpark, para analizar grandes volúmenes de datos y proporcionar insights sobre las ventas efectuadas en el comercio electrónico.

## B. Realizar pruebas de funcionamiento de los componentes implementados

Se realizaron pruebas de los componentes descargando los archivos necesarios del repositorio de GitHub del proyecto en un entorno local, para luego desplegar los clústeres de contenedores y de datos distribuidos. Durante estas pruebas, se verificaron las funcionalidades que debían ofrecer los microservicios a cada tipo de usuario de la aplicación, asegurando que todas las operaciones CRUD (Crear, Leer, Actualizar y Borrar) se ejecutaran correctamente. Para esto, se revisaron los contenedores de las bases de datos y sus servicios de MySQL, garantizando que la gestión de datos se realizara de manera eficiente y sin errores.

En el caso del procesamiento de datos distribuido, se empleó un conector de MySQL en Spark para comunicarse con las bases de datos de productos y carritos, creando DataFrames que luego fueron analizados utilizando PySpark. Usando el entorno de Spark instalado y configurado localmente, se ejecutó el algoritmo análisis.py para generar visualizaciones y estadísticas sobre las ventas. Se verificó que las gráficas generadas se guardaran en una carpeta compartida entre los dos nodos de procesamiento, que actuaba como volumen para los contenedores de servicios web, y se comprobó que se reflejaran correctamente en ellos. Finalmente, se verificó que estas imágenes estuvieran accesibles en la pestaña de estadísticas del usuario administrador, mostrando un dashboard con los resultados del análisis.

## C. Pruebas

Para evaluar el rendimiento de la aplicación se utilizó Apache JMeter como herramienta principal. Las pruebas realizadas buscan identificar cómo responde la aplicación bajo diferentes niveles de carga y estrés, definiendo tres escenarios clave:

* **Simulación de carga normal:** este caso consiste en modelar un tráfico estable, empleando 100 usuarios durante 3 ciclos y un lapso de 30 segundos. La intención es medir el rendimiento de la aplicación en condiciones de uso cotidiano.
* **Simulación de alta carga**: aquí se incrementa significativamente la carga, con 500 usuarios que interactúan en 5 ciclos. Este escenario permite analizar el comportamiento del sistema cuando se enfrenta a demandas mayores.
* **Prueba de estrés:** en esta evaluación se busca llevar al sistema a su límite, sometiéndolo a 1000 usuarios durante 10 ciclos. Esto permite identificar el punto en el que la aplicación comienza a degradarse o fallar debido a la sobrecarga.

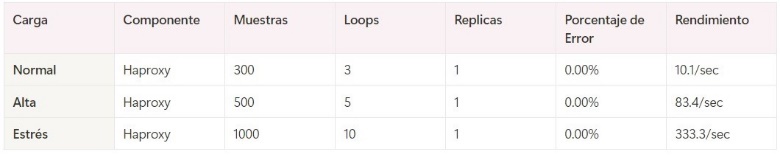
Para evaluar el rendimiento, se analizaron dos métricas fundamentales:

* **Promedio de tiempo de respuesta (Rendimiento):** Representa la cantidad de operaciones exitosas completadas por unidad de tiempo, indicando el rendimiento general [21].
* **Tasa de errores:** Calcula el porcentaje de acciones fallidas durante la prueba, permitiendo identificar problemas en la estabilidad del sistema [21].

Los resultados obtenidos se evidencian en la siguiente tabla.

TABLA IV

Escenarios de prueba y resultados de métricas evaluadas



Referencias

1. N. Singh et al., “Load balancing and service discovery using Docker Swarm for microservice based big data applications,” Journal of Cloud Computing, vol. 12, no. 1, Jan. 2023, doi: https://doi.org/10.1186/s13677-022-00358-7.
2. U. K. Baliyarsingh, Jul. 2024. “Retail Sales and Customer Behavior Analysis”, Kaggle.com, https://www.kaggle.com/datasets/utkalk/large-retail-data-set-for-eda
3. N. López, E. Erazo, M. Burgos and J.D. Daza. (2024, Sep. 19). *Proyecto ReI (parte 1)* [Online]. Available: https://docs.google.com/document/d/16Ipq-0NrEebLbKkUQV2xzo1WvITF9R4UHpgX-u3pF6Y/edit?usp=sharing
4. S. Mazhar, J. Copeland, and S. Abu-Naser, "Docker Swarm and Kubernetes for Containers Orchestration: A Performance Evaluation," 2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), New York, NY, USA, 2019, pp. 1-5. doi: 10.1109/UEMCON47517.2019.8993083.
5. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," ACM Queue, vol. 14, no. 1, pp. 70-93, Jan. 2016. doi: 10.1145/2898442.2898444.
6. S. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, no. 3, pp. 42-52, May 2016. doi: 10.1109/MS.2016.64.
7. B. Hindman et al., "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2011, pp. 295-308. doi: 10.5555/1972457.1972488.
8. M. Zaharia et al., "Apache Spark: A Unified Engine for Big Data Processing," Communications of the ACM, vol. 59, no. 11, pp. 56-65, Nov. 2016, doi: 10.1145/2934664.
9. M. Rocklin, "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling," in Proceedings of the 14th Python in Science Conference, Austin, TX, USA, 2015, pp. 126-132.
10. S. Ewen, K. Tzoumas, S. Markl, and D. Warneke, "Apache Flink: Stream and Batch Processing in a Single Engine," Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 38, no. 4, pp. 28-38, Dec. 2015. DOI: 10.1109/ICDE.2015.7113333.
11. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, vol. 51, no. 1, pp. 107-113, Jan. 2008, doi: 10.1145/1327452.1327492.
12. IBM, (2023). *What is Three-Tier Architecture*. [Online]. Available: https://www.ibm.com/topics/three-tier-architecture
13. G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou and Z. Li, "Microservices: architecture, container, and challenges," *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Macau, China, 2020, pp. 629-635, doi: 10.1109/QRS-C51114.2020.00107.
14. C. Pahl, “Containerization and the PaaS Cloud,” IEEE Cloud Computing, vol. 2, no. 3, pp. 24–31, May 2015, doi: http://dx.doi.org/10.1109/MCC.2015.51
15. (2024). *Whats is a Container?* [Online]. Available: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/
16. RedHat. (2022, May. 10). *What is container orchestration?* [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration
17. Docker Inc. (2024, Oct. 24). *Deploy to swarm* [Online]. Available: https://docs.docker.com/guides/swarm-deploy/
18. Docker Inc. (2024, Sep. 10). *How nodes work* [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/
19. S. S, “Building Scalable Data Pipelines with Apache Kafka and Spark,” Medium, Dec. 15, 2023. [Online]. Available: https://medium.com/@shreyanth98/building-scalable-data-pipelines-with-apache-kafka-and-spark-ccfb8b72a83f
20. M. Chojnowska. (2023, Apr. 26). *Building a Scalable Apache Spark Cluster - A Beginner's Guide* [Online]. Available: https://sunscrapers.com/blog/building-a-scalable-apache-spark-cluster-beginner-guide/
21. Apache Software Foundation. (2024). *Apache JMeter - User’s Manual: Glossary* [Online]. Available: https://jmeter.apache.org/usermanual/glossary.html

1. Michel Dahiana Burgos Santos is with the Data and Artificial Intelligence Engineering Undergraduate Program, Universidad Autónoma de Occidente, Cali, Valle del Cauca 760030 Colombia (e-mail: michel.burgos@uao.edu.co).

   Juan David Daza Rivera is with the Data and Artificial Intelligence Engineering Undergraduate Program, Universidad Autónoma de Occidente, Cali, VAC 760030 CO (e-mail: juan\_dav.daza@uao.edu.co).

   Liseth Esmeralda Erazo Varela is with the Data and Artificial Intelligence Engineering Undergraduate Program, Universidad Autónoma de Occidente, Cali, VAC 760030 CO (e-mail: liseth.erazo@uao.edu.co).

   Natalia López Gallego is with the Data and Artificial Intelligence Engineering Undergraduate Program, Universidad Autónoma de Occidente, Cali, VAC 760030 CO (e-mail: natalia.lopez\_gal@uao.edu.co). [↑](#footnote-ref-1)