Unidad de Planeación
Minero Energética

INFORME
LINEAMIENTOS DE ARQUITECTURA DE SOFTWARE

SUBDIRECCIÓN DE GESTIÓN DE LA INFORMACIÓN

CIO SECTORIAL
CHIEF INFORMATION OFFICER

UPME
Av Calle 26 # 69D – 91 Torre 1 – Piso 9
Bogotá – Colombia
Tel.: +57 6012220601
upme.gov.co

UNIDAD DE PLANEACIÓN MINERO ENERGÉTICA - UPME

MANUEL PEÑA SUÁREZ
Director General (E)

JOHANNA STELLA CASTELLANOS
Subdirectora Gestión de la Información

Equipo Subdirección Gestión de la Información

JUAN ANTONI DOMÍNGUEZ
RAFAEL SANTOS ARNEDO MENDOZA
FABIAN GARZÓN GARCÍA

República de Colombia - Ministerio de Minas y Energía
Septiembre 2025

TABLA DE CONTENIDO

INTRODUCCIÓN

En el desarrollo de sistemas modernos, la arquitectura de software juega un papel fundamental para garantizar la escalabilidad, mantenibilidad, seguridad, resiliencia y capacidad de adaptación frente a los constantes cambios y necesidades del negocio. A medida que las organizaciones buscan ofrecer soluciones digitales más ágiles y competitivas, se enfrentan a desafíos como la integración de múltiples servicios, la orquestación de flujos de datos en tiempo real, el soporte de altos volúmenes de información y la interoperabilidad entre diferentes plataformas y tecnologías.

En este contexto, resulta indispensable adoptar enfoques arquitectónicos que vayan más allá del diseño monolítico tradicional, impulsando modelos distribuidos, desacoplados y orientados al dominio del negocio. Conceptos como Domain Driven Design (DDD) permiten que la lógica del software esté alineada con el lenguaje y procesos del negocio, reduciendo la complejidad y facilitando la evolución de los sistemas. Asimismo, las arquitecturas distribuidas y los brokers de eventos promueven la comunicación eficiente y asíncrona entre componentes, lo cual es clave para aplicaciones que requieren alta disponibilidad y procesamiento en tiempo real.

De igual manera, el uso de microservicios y APIs posibilita el desarrollo modular y escalable de aplicaciones, promoviendo la independencia de los equipos de trabajo y favoreciendo la integración continua. Finalmente, el despliegue mediante contenedores asegura portabilidad, consistencia en los entornos de ejecución y eficiencia en la administración de recursos.

Este informe recopila y desarrolla lineamientos clave en torno a Domain Driven Design (DDD), arquitecturas distribuidas, brokers de eventos, microservicios, APIs y contenedores, con un enfoque aplicado al desarrollo de

1. MARCO CONCEPTUAL

En un equipo colaborativo como la UPME, constantemente los desarrollos son realizados y compartidos con los interesados ya sea mediante un repositorio o exponiendo un servicio para ser consumido. En el contexto de trabajo equipo, es necesario la estandarización en la codificación y definición del patrón de diseño para que todo aquel involucrado dentro del proyecto pueda trabajar en cualquier implementación sin requerir un esfuerzo adicional.

2. DOMAIN DRIVEN DESIGN (DDD)

DDD propone alinear la arquitectura del software con el dominio del negocio. Esto permite que el diseño del sistema refleje con precisión los procesos, reglas y entidades relevantes.
Principios Clave:

- Lenguaje Ubicuo (Ubiquitous Language): Todos los actores (desarrolladores, analistas, stakeholders) deben hablar un mismo lenguaje basado en el dominio.

- Entidades y Value Objects:
  - Entidades tienen identidad propia en el tiempo.
  - Value Objects representan atributos inmutables sin identidad
- Agregados y Repositorios:
  - Los Agregados agrupan entidades y definen reglas de consistencia.
  - Los Repositorios gestionan la persistencia.
- Bounded Contexts: Cada subdominio tiene su propio contexto delimitado con responsabilidades claras.

Estos atributos permiten la modularización de la API y facilitar los cambios de dependencias de forma rápida además de:

- Separar la aplicación en módulos/layers siguiendo el contexto delimitado.
- Definir models (Pydantic + SQLAlchemy) representando Entidades y VO.
- Usar services para reglas de negocio y repositories para persistencia.

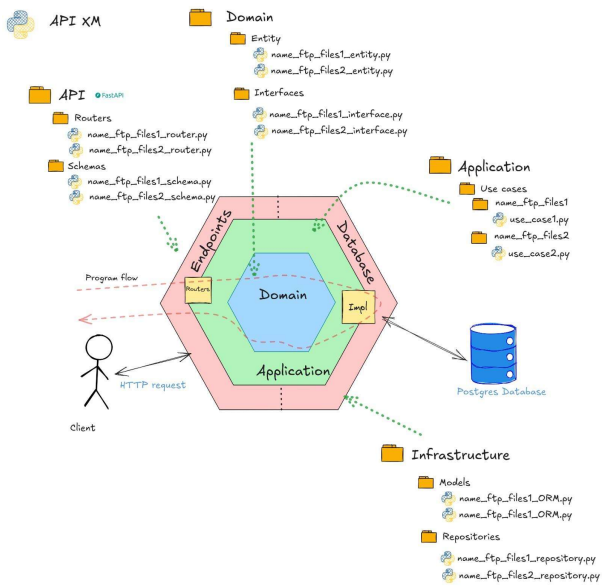

Figura 1: Capas de metodología Domain Driven Design

Capas de la Arquitectura DDD

Aunque el DDD no impone una arquitectura de capas estricta, la arquitectura clásica que se utiliza para implementarlo, conocida como Arquitectura Hexagonal o de Puertos y Adaptadores, organiza el software en capas concéntricas que protegen el núcleo del dominio.
Las capas principales de esta metodología son:

❖ Capa del Dominio (Domain Layer)

Es el corazón del sistema y la más importante. Contiene toda la lógica de negocio, reglas y el modelo del dominio (la representación de la realidad del negocio). No tiene dependencias de otras capas.

- Objetos de Dominio: Entidades, objetos de valor, agregados.
- Servicios de Dominio: Lógica que no encaja en una entidad o un objeto de valor.
- Eventos de Dominio: Notificaciones de que algo importante ocurrió en el dominio.

❖ Capa de Aplicación (Application Layer)

Coordina las tareas y flujos de trabajo de la aplicación. Es la que dirige el orquestamiento de los objetos del dominio para que ejecuten la lógica de negocio. Es delgada y no contiene lógica de negocio, solo coordina.

- Servicios de Aplicación: Operaciones que un cliente puede solicitar (ej. "crear un nuevo pedido", "cambiar la dirección de un cliente").
- Transacciones y Seguridad: Maneja la gestión de transacciones y la autorización.

❖ Capa de Infraestructura (Infrastructure Layer)

Se encarga de los detalles técnicos que soportan a las otras capas. Esta capa es la que permite que las aplicaciones se comuniquen con el mundo exterior.

- Persistencia: Repositorios que guardan y recuperan datos de la base de datos.
- Frameworks de UI: Implementa la interfaz de usuario.
- Comunicación Externa: Se encarga de la comunicación con APIs, sistemas de mensajería, etc.
- Librerías de Utilidad: Clases para logging, seguridad, etc.

❖ Capa de Interfaz de Usuario / Presentación (User Interface / Presentation Layer)

Esta capa es responsable de mostrar la información al usuario y traducir sus comandos a la capa de aplicación. Se preocupa por la interacción con el usuario.

- Controladores: Clases que reciben peticiones del usuario (por ejemplo, en una API REST o una interfaz web).
- Vistas: Componentes visuales que presentan la información.
- Mapeo de Datos: Traduce la información del dominio a un formato que la interfaz de usuario pueda entender.

3. ARQUITECTURAS DISTRIBUIDAS

Las arquitecturas distribuidas permiten que múltiples servicios se comuniquen entre sí a través de la red, logrando escalabilidad horizontal y resiliencia.

Características:

- Desacoplamiento: Cada servicio funciona de manera independiente.
- Escalabilidad: Posibilidad de replicar servicios críticos según la demanda.
- Tolerancia a fallos: Un fallo en un servicio no detiene todo el sistema.
- Comunicación: Puede ser síncrona (APIs REST/gRPC) o asíncrona (eventos).

Retos:

- Consistencia de datos: Uso de patrones como Saga Pattern o Event Sourcing.
- Monitoreo y observabilidad: Necesario implementar logs distribuidos, métricas y tracing.

4. BROKERS DE EVENTOS

Los brokers de eventos (ej. Redis , Kafka) permiten comunicación asíncrona entre servicios, siendo fundamentales para arquitecturas reactivas.
Beneficios:

- Desacoplamiento temporal: Los productores no dependen de la disponibilidad de los consumidores.
- Escalabilidad: Permite procesar grandes volúmenes de mensajes.
- Event-driven Architecture (EDA): Los eventos del negocio impulsan las acciones.
-

Aplicación con FastAPI:

- Integrar librerías como aiokafka o faststream para publicar/consumir eventos.
- Definir esquemas de eventos con Pydantic para garantizar validación y consistencia.

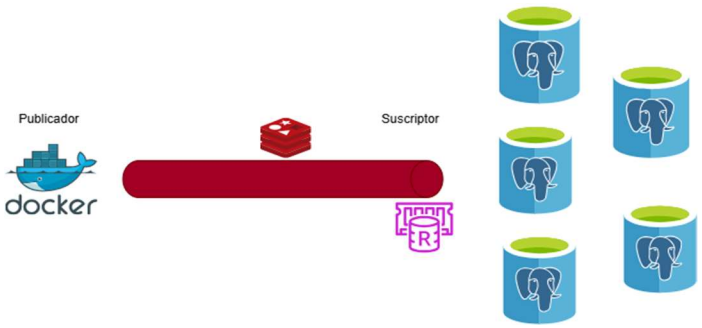

Figura N: Uso de brokers para él envió de información.

5. MICROSERVICIOS

Los microservicios representan la descomposición de una aplicación monolítica en servicios pequeños, autónomos y desplegables de forma independiente.
Características:

- Autonomía: Cada microservicio puede desarrollarse, desplegarse y escalar sin afectar a otros.
- Responsabilidad única: Un microservicio resuelve un problema específico del dominio.
- Tecnologías heterogéneas: Cada servicio puede implementarse con el stack más conveniente.
- Comunicación:
    - Síncrona: REST/gRPC.
    - Asíncrona: Brokers de eventos.

Con FastAPI:

- Se puede implementar cada microservicio como una API independiente, ligera y performante.
- Uso de dependency injection y middlewares para seguridad y validación.

6. APIs

Las APIs son el contrato de comunicación entre servicios y clientes.
Tipos principales:

- REST: Basado en recursos y HTTP (GET, POST, PUT, DELETE).
- GraphQL: Consultas flexibles, el cliente decide qué información obtener.
- gRPC: Comunicación binaria de alto rendimiento.

Buenas prácticas:

- Versionar las APIs (/api/v1/...).
- Documentar automáticamente (FastAPI lo hace con OpenAPI/Swagger).
- Implementar seguridad (OAuth2, JWT, API Keys).
- Manejar rate limiting y caching.

7. CONTENEDORES Y ORQUESTADORES

La arquitectura de microservicios ha transformado el desarrollo de software al descomponer aplicaciones monolíticas en servicios pequeños, independientes y acoplados de forma flexible. Esta descentralización, si bien ofrece una agilidad sin precedentes y una mayor resiliencia, presenta un desafío significativo en su gestión a gran escala. La proliferación de estos servicios, cada uno encapsulado en su propio contenedor, hace que el control manual de su ciclo de vida sea inviable. La orquestación de contenedores emerge como la solución indispensable para este problema, proporcionando una capa de automatización que gestiona el despliegue, la configuración, la escalabilidad y la tolerancia a fallos.

Uno de los más conocidos gestionadores de contenedores y de imágenes es Docker. Docker permite el despliegue de servicios y la gestión de imágenes de forma rápida, lanzando aplicaciones con una sola instrucción. Estas aplicaciones pueden ser configuradas por el desarrollador donde este le instala sus dependencias y la aísla de otros contextos dentro del mismo host. Esto representa un gran ventaja ya que todas las dependencias se encuentran en la imagen docker que puede ser subidas a algún docker registry y ser reutilizada.

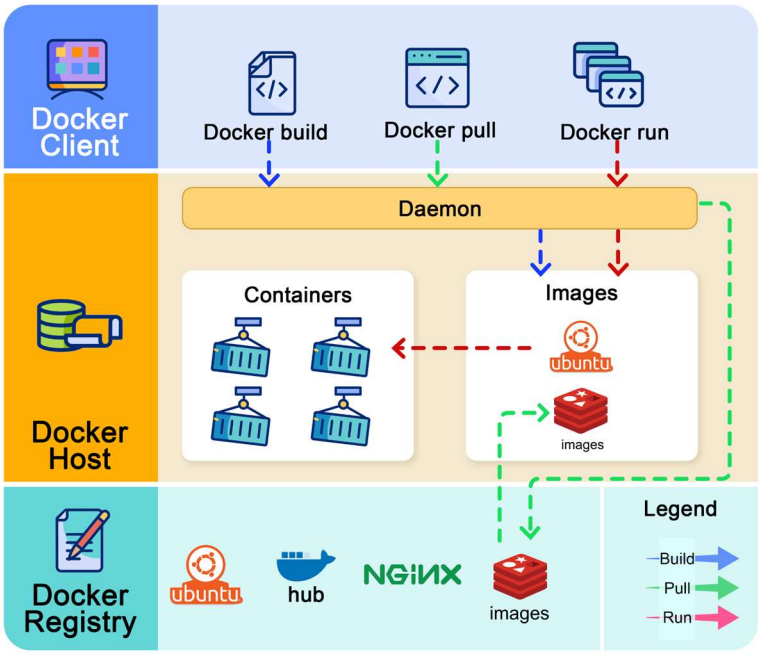

Figura N. Funcionamiento Docker

La figura N. Muestra el funcionamiento simplificado de Docker, el docker daemon recibe la instrucción y se encargan de desplegar el servicio según las instrucciones dadas. Las imágenes son tomadas

desde un docker registry y se reutilizan dentro del host, estas imágenes construidas por terceros pueden ser utilizadas para lanzar servicios como Redis, Postgres, Airflow, Nginx entre muchos más. E

También podemos construir imágenes custom como incluir un script de python que realiza un pipeline ETL, como la figura de N2.

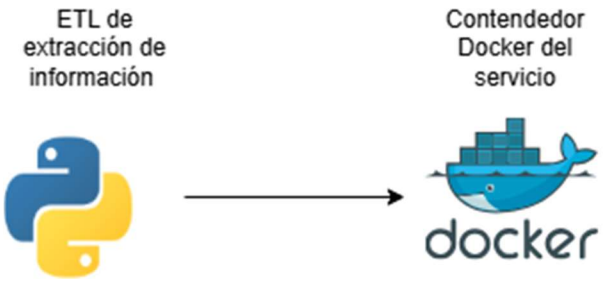

Figura N2. ETL en python como microservicio

En una arquitectura de microservicios, cada servicio se ejecuta idealmente en su propio contenedor. Por ejemplo, una aplicación construida con FastAPI, que expone una API REST para un servicio específico, puede ser empaquetada en una imagen de Docker y ejecutada en un contenedor, lo que proporciona un aislamiento de recursos y una portabilidad completa. Esta granularidad permite que cada servicio se desarrolle, se escale y se despliegue de forma independiente, lo que es un pilar de la agilidad en los equipos de desarrollo.

Esto trae diferentes retos como:

- **Gestión de Fallos:** Si un contenedor o el servidor que lo aloja falla, ¿cómo se detecta el problema y se reinicia la carga de trabajo automáticamente para asegurar la continuidad del servicio?
- **Escalabilidad:** ¿Cómo se ajusta el número de réplicas de una aplicación para manejar picos de tráfico sin intervención manual?
- **Descubrimiento de Servicios:** ¿Cómo se comunican los microservicios entre sí si sus direcciones IP son efímeras y cambian con cada reinicio?
- **Balanceo de Carga:** ¿Cómo se distribuye el tráfico entrante de manera uniforme entre todas las réplicas de un servicio para evitar la sobrecarga de un solo contenedor?

Kubernetes

La arquitectura de un clúster de Kubernetes se divide conceptualmente en dos capas principales que trabajan en un bucle de control continuo para garantizar que el estado del clúster coincida con la configuración deseada. Estas capas son el Plano de Control (Control Plane) y los Nodos de Trabajo (Worker Nodes).

El Plano de Control

El Plano de Control es la capa de gestión que toma las decisiones y mantiene el estado deseado del clúster. No ejecuta directamente las cargas de trabajo del usuario, sino que orquesta los componentes que lo hacen. La figura N3 muestra la arquitectura de Kubernetes. Está compuesto por varios procesos clave:

- **kube-apiserver**: Es el componente central y la única puerta de entrada para todas las comunicaciones del clúster. Actúa como un intermediario entre las peticiones de los usuarios (por ejemplo, a través de la herramienta de línea de comandos kubectl) y los componentes del clúster. El apiserver valida y configura los objetos de la API de Kubernetes, como **Pods** y **Deployments**, y es el único proceso que se comunica con el almacén de estado del clúster.
- **etcd**: Es un almacén de datos distribuido y de alta disponibilidad que funciona como un sistema de almacenamiento de clave-valor. Es el "único punto de verdad" del clúster, donde se guarda de forma persistente toda la información de estado, las configuraciones, los secretos y la información de los nodos. Su robustez y consistencia son críticas para la integridad del clúster.
- **kube-scheduler**: Este componente es responsable de asignar los Pods recién creados a los Nodos de Trabajo. El **scheduler** tiene en cuenta una variedad de factores en su toma de decisiones, incluyendo la disponibilidad de recursos (CPU, memoria), las reglas de afinidad y antiafinidad, y la distribución de Pods para evitar la sobrecarga de un solo nodo.
- **kube-controller-manager**: Un proceso monolítico que ejecuta múltiples controladores en bucles de control continuos. Cada controlador se encarga de un aspecto específico del clúster. Por ejemplo, el **Node Controller** monitorea la salud de los nodos y toma medidas cuando uno falla, mientras que el **Replication Controller** asegura que el número deseado de réplicas de un Pod siempre esté en ejecución.

Nodos

Los Nodos de Trabajo son las máquinas físicas o virtuales donde se ejecutan las cargas de trabajo del usuario. Cada nodo es gestionado por el Plano de Control y contiene los siguientes componentes esenciales:

- **kubelet**: Un agente que se ejecuta en cada nodo y es el enlace principal con el Plano de Control. El kubelet observa el kube-apiserver para detectar nuevos Pods que han sido asignados a su nodo. Una vez que un Pod es asignado, el kubelet interactúa con el *Container Runtime* local para asegurar que los contenedores del Pod se estén ejecutando y mantengan su estado de salud.

- **kube-proxy**: Un *proxy* de red que se ejecuta en cada nodo. Mantiene las reglas de red en el nodo para facilitar la comunicación entre los Pods y con el exterior del clúster, permitiendo el balanceo de carga interna.
- **Container Runtime**: El software subyacente responsable de ejecutar los contenedores, como containerd (el más utilizado), CRI-O o el Docker Engine.

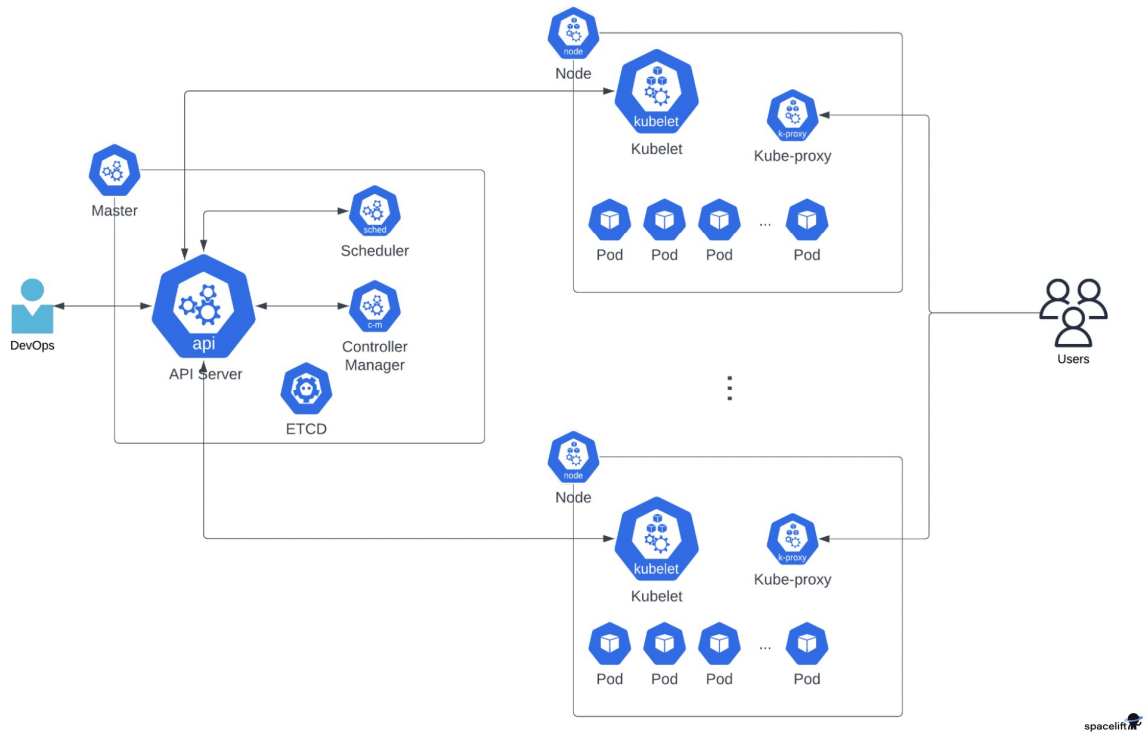

Figura N3. Arquitectura de Kubernetes (Tomado de SpaceLift)

Automatización con GitLab CI/CD

La automatización del despliegue en Kubernetes es el paso final para un flujo de trabajo eficiente. GitLab CI/CD se integra de manera segura con los clústeres de Kubernetes a través del GitLab Agent for Kubernetes.

- **Integración Segura**: A diferencia de los métodos más antiguos basados en certificados (más propensos a vulnerabilidades), el GitLab Agent se instala en el clúster y mantiene una conexión segura con GitLab, permitiendo a los pipelines interactuar con el clúster sin necesidad de exponer los endpoints de la API. Este enfoque es un modelo de "acceso de

extracción" (pull-based) donde el agente toma la configuración, en lugar del modelo "acceso de empuje" (push-based) del pasado.

- **Flujo del Pipeline**: El archivo .gitlab-ci.yml define las etapas y los trabajos del pipeline. Para el despliegue de una aplicación FastAPI, un pipeline típico podría incluir las siguientes etapas :
    1. **build**: Construir la imagen de Docker a partir del Dockerfile optimizado.
    2. **test**: Ejecutar pruebas unitarias y de integración.
    3. **deploy**: En esta etapa, el pipeline utiliza el GitLab Agent para obtener el contexto del clúster de Kubernetes y ejecutar comandos kubectl. Por ejemplo, kubectl apply -f deployment.yaml actualiza el Deployment con la nueva versión de la imagen, iniciando el rolling update de forma automatizada y controlada.

8. LINEAMIENTOS GENERALES

Teniendo en cuenta lo anteriormente descrito, se proponen los siguientes lineamientos de buenas prácticas para el diseño y desarrollo de arquitecturas de sistemas de información:

A. Diseñar desde el dominio (DDD): Adoptar el enfoque de *Domain-Driven Design* para modelar el sistema a partir del conocimiento del dominio, facilitando así la alineación entre los requisitos del negocio y la solución técnica.
B. Adoptar microservicios en contextos apropiados: Implementar una arquitectura de microservicios cuando se enfrente una alta complejidad en el dominio o se trabaje con equipos de desarrollo grandes y distribuidos.
C. Usar brokers de eventos para flujos asíncronos: Emplear herramientas de mensajería como Kafka o RabbitMQ para orquestar la comunicación asíncrona entre servicios, mejorando la escalabilidad y desacoplamiento del sistema.
D. Exponer APIs estandarizadas, documentadas y seguras: Diseñar interfaces de programación de aplicaciones (APIs) siguiendo estándares como REST o GraphQL, asegurando su documentación (por ejemplo, con OpenAPI/Swagger) y protegiéndolas mediante autenticación y autorización.
E. Contenerizar y orquestar los servicios: Utilizar tecnologías como Docker para contenerizar los servicios y Kubernetes para su orquestación, lo que facilita la escalabilidad, el despliegue y la gestión en entornos de producción.
F. Implementar observabilidad: Incorporar herramientas de monitoreo y trazabilidad como Prometheus, Grafana y la pila ELK (Elasticsearch, Logstash, Kibana) para obtener visibilidad sobre el comportamiento del sistema y facilitar la detección de errores o cuellos de botella.
G. Automatizar CI/CD con pipelines: Establecer procesos automatizados de integración y entrega continua (CI/CD), mediante pipelines que aseguren la calidad del software, reduzcan errores humanos y agilicen los despliegues.

Unidad de Planeación
Minero Energética

www.upme.gov.co