Distributed Machine Learning (DML):

Introducción

DML: es un sistema que permite a los usuarios entrenar modelos de machine learning de forma distribuida y realizar predicciones utilizando modelos previamente entrenados.

El sistema expone una **API REST** que ofrece los servicios de entrenamiento y predicción, y además incluye una aplicación de consola que encapsula toda la lógica necesaria para que el funcionamiento del sistema sea completamente transparente para el usuario.

Alcance del Sistema

1. El sistema ofrece diversos modelos de machine learning para tareas de ***regresión*** y ***clasificación.***
2. Permite a los usuarios subir archivos **.csv** con los datasets de entrenamiento o predicción.
3. Los usuarios pueden crear procesos de entrenamiento indicando:
   - el tipo de entrenamiento (regresión o clasificación).
   - el dataset que se usará.
   - los modelos a entrenar.
4. El sistema permite consultar el estado de los entrenamientos en curso.
5. Los usuarios pueden ejecutar predicciones utilizando modelos previamente entrenados.
6. Se pueden consultar los resultados generados por las predicciones.
7. Los usuarios pueden descargar los modelos ya entrenados.

Especificaciones del Sistema:

El sistema utiliza una arquitectura modular y desacoplada basada en dos tipos de nodos (Workers y DataBases)

Workers:

Los nodos Worker son responsables de procesar las peticiones de los usuarios y ejecutar las tareas de entrenamiento y predicción de los modelos. Cada Worker es una instancia del backend (`dml-service`) que expone una API REST construida con FastAPI.

Cada Worker funciona de forma totalmente independiente, sin necesidad de comunicarse ni coordinarse con otros Workers. Su única interacción es con los

nodos DataBase, que actúan como fuente de datos y como mecanismo de control del trabajo distribuido.

Procesos de los Workers:

Cada Worker ejecuta 4 tipos de procesos distintos:

1. REST (API Endpoints):

Procesos que manejan las peticiones HTTP a través de los endpoints definidos en `app/api/endpoints/`. Se encargan de:

- **`/training/train`**: Crear sesiones de entrenamiento, generar `training_id` único e inicializar modelos en background
- **`/predictions/predict`**: Registrar sesiones de predicción para un modelo y dataset
- **`/datasets`**: Gestionar datasets disponibles
- **`/model_registry`**: Consultar información de modelos
- **`/health`**: Verificar el estado del Worker
- **`/leader/*`**: Endpoints para elección de líder entre Workers (`/compare`, `/announce`, `/get`, `/start-election`)

Estos endpoints procesan las peticiones de los clientes, validan los datos de entrada mediante esquemas Pydantic (`app/schemas/`), y realizan las operaciones necesarias sobre los nodos DataBase a través del `DatabaseManager`.

2. GET_MODEL (Polling de Tareas):

Proceso implementado en la clase `Runner` (`app/models/runner.py`) que ejecuta un hilo de polling continuo mediante el método `_poll_for_models()`. Su funcionamiento:

1. Verifica si hay capacidad disponible (controlado por `MAX_CONCURRENT_RUNNING_MODELS`)
2. Llama a `database_manager.get_model_to_run()` para obtener modelos disponibles
3. Recibe un objeto `ModelToRun` con:
   - **`model_id`**: Identificador del modelo
   - **`running_type`**: Tipo de tarea (`training` o `prediction`)
   - **`dataset_id`**: Dataset asociado
4. Inicia la ejecución del modelo mediante `_start_model_execution()`
5. Limpia modelos completados con `_cleanup_completed_models()`

El polling se ejecuta cada **2 segundos**.

3. Training/Prediction (Ejecución de ML):

Procesos gestionados por la clase `Runner` que crea dos hilos por cada modelo:

Hilo de Ejecución (`_run_model`): - Obtiene los datos completos del modelo desde el DataBase - Instancia la clase del modelo correspondiente (ej: `RandomForestClassifierModel`) - Deserializa el estado previo del modelo - Ejecuta `_execute_training()` o `_execute_prediction()` según el `running_type` - Serializa y guarda los resultados

Hilo de Status (`_send_status_notifications`): - Envía heartbeats cada **10 segundos** mediante `database_manager.send_model_running_status()` - Se detiene cuando el hilo de ejecución termina

Las clases de modelos heredan de `MLModel` (`app/models/ml_model.py`) e implementan los métodos `train()`, `predict()`, `serialize()` y `deserialize()`.

4. Descubrimiento (Discovery):

Hilo implementado en la clase `Middleware` (`app/middleware/middleware.py`) mediante el método `_discover_ips()`:

- Se inicia con `start_monitoring()` al arrancar la aplicación
- Ejecuta `_refresh_service_ip_cache()` cada **10 segundos**
- Resuelve IPs mediante DNS con `_resolve_domain_ips()`
- Mantiene un `ip_cache` con todos los nodos descubiertos
- Verifica la salud de cada IP con `check_ip_alive()` haciendo peticiones a `/api/v1/health`
- Elimina IPs que no responden del caché
- Usa `cache_lock` (threading.Lock) para acceso thread-safe al caché

Tolerancia a Fallas de los Workers:

Los Workers están diseñados con total independencia:

- **No conocen** la existencia de otros Workers
- **No comparten estado** ni intercambian mensajes entre sí
- **No requieren coordinación** directa

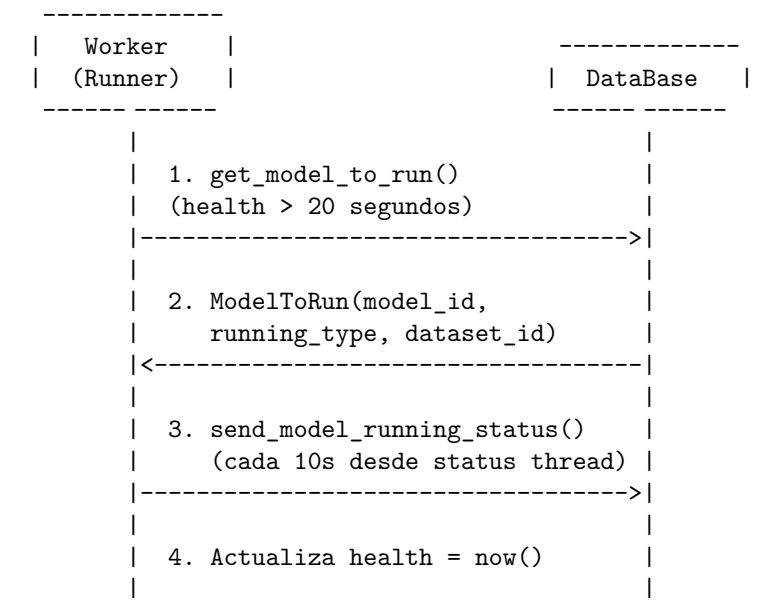
Garantías:

Característica	Descripción
Escalabilidad Horizontal	Agregar nuevos Workers sin cambios de configuración
Tolerancia a Fallas	K-1, donde K es la cantidad de Workers activos
Resistencia a Particiones	El sistema funciona mientras exista al menos un Worker activo
Recuperación Automática	Workers pueden reconectarse sin intervención manual

Coordinación entre Workers:

Para evitar que dos Workers procesen el mismo modelo simultáneamente, el sistema utiliza un mecanismo de **heartbeat** basado en el campo `health` almacenado en los nodos DataBase.

Funcionamiento del Mecanismo:



Etapas del Proceso:

1. **Selección del modelo disponible**:
   - Los nodos DataBase registran el `health` (timestamp) de cada modelo
   - El método `get_model_to_run()` retorna modelos cuyo `health` sea mayor a **20 segundos** (modelo libre o abandonado)
2. **Heartbeat del Worker**:
   - El hilo `_send_status_notifications()` envía señales cada **10 segundos**
   - Llama a `database_manager.send_model_running_status(model_id, dataset_id)`
   - El DataBase actualiza: `health = timestamp_actual`
3. **Detección de fallo o abandono**:
   - Si no se recibe heartbeat en más de **20 segundos**, el modelo se considera abandonado
   - El modelo queda disponible para reasignación
4. **Reasignación automática**:
   - Cualquier Worker puede tomar un modelo libre inmediatamente
   - No requiere bloqueos distribuidos ni coordinación entre Workers

Ventajas del Mecanismo:

Ventaja	Descripción
Sin conflictos	Ningún modelo será procesado por dos Workers simultáneamente
Alta disponibilidad	Si un Worker falla, el trabajo se reasigna automáticamente
Escalabilidad simple	Más Workers = más capacidad sin cambios estructurales
Desacoplamiento total	La lógica distribuida se delega a los nodos DataBase
Sin bloqueos distribuidos	El campo `health` elimina la necesidad de locks complejos

DataBases:

Los nodos DataBase emplean una arquitectura peer to peer (p2p), comunicándose entre ellos, sin necesidad de un líder que controle el flujo de datos.

Cada DataBase expone una API REST para responder a las solicitudes que reciben desde los Workers o el resto de DataBases.

Cada nodo mantiene su propio almacenamiento, procesa consultas de lectura y escritura, y participa en el mecanismo de replicación distribuida para lograr la consistencia del sistema.

Procesos

Cada nodo DataBase ejecuta varios procesos concurrentes para mantener el sistema funcionando:

1. REST API (Servidor Principal)

Expone endpoints para interactuar con Workers y otros nodos DataBase: - **`/datasets`**: Subida (con particionado en batches) y descarga de datasets. - **`/models`**: Registro de modelos, consulta de tareas pendientes (`/torun`), y actualizaciones de estado (`/health`). - **`/predictions`**: Guardado y consulta de predicciones. - **`/peers`**: Endpoint interno para sincronización de metadatos entre nodos DB. - **`/trainings`**: Creación y gestión de entrenamientos.

2. Middleware de Descubrimiento (`_discover_ips`)

- **Frecuencia**: Cada 10 segundos.
- **Función**: Resuelve el nombre de dominio del servicio (DNS de Docker) para encontrar nuevas IPs de nodos DataBase y actualiza una caché local. Verifica la salud de los nodos conocidos mediante peticiones HTTP a `/health`.

3. **Sincronización de Metadatos (`_sync_loop`)**

- **Frecuencia**: Cada 7-10 segundos.
- **Función**: Intercambia el objeto `PeerMetadata` con otros nodos sanos. Esto permite que cada nodo sepa qué datos (datasets, modelos) posee cada par en la red.
- **Lógica de Merge**: Al recibir metadatos de otro par, se fusionan las listas de nodos poseedores y se actualizan los registros CSV locales si el par tiene información más reciente.

4. **Monitor de Replicación (`_replication_monitor`)**

- **Frecuencia**: Cada 20 segundos.
- **Función**: Verifica si los recursos (datasets, modelos, predicciones) cumplen con el `REPLICATION_FACTOR` (por defecto 3).
- **Coordinación**: Si un recurso está sub-replicado, **solo el nodo poseedor con la IP más baja** inicia la replicación hacia otros nodos disponibles. Esto evita que múltiples nodos intenten replicar el mismo dato simultáneamente.

5. **Persistencia de Metadatos (`_persist_loop`)**

- **Frecuencia**: Cada 5 segundos.
- **Función**: Guarda el estado actual de `PeerMetadata` en disco (`peer_metadata.json`) para recuperar el conocimiento de la red en caso de reinicio.

Modelos de Datos

El sistema utiliza una combinación de archivos CSV para metadatos y archivos JSON/CSV para datos voluminosos:

Datasets

- **Metadatos**: `datasets.csv` (ID, número de batches).
- **Datos**: Archivos CSV particionados: `{dataset_id}_batch_{n}.csv`.
- **Almacenamiento**: Carpeta `data/datasets/`.

Modelos

- **Metadatos**: `models.csv`. Contiene el estado global y métricas.
  - Campos clave: `model_id`, `status` (pending, training, completed), `health` (timestamp), `metrics...`.
- **Datos**: Archivos JSON serializados: `{model_id}.json`. Contiene los parámetros del modelo entrenado.
- **Almacenamiento**: Carpeta `data/models/`.

Entrenamientos

- **Metadatos**: `trainings.csv`. Agrupa múltiples modelos bajo un mismo experimento.
  - Campos: `training_id`, `dataset_id`, `type`, `status`.

Predicciones

- **Metadatos**: Se registran en `models.csv` con `task="prediction"`.
- **Datos**: Archivos JSON: `{model_id}_{dataset_id}.json`.
- **Almacenamiento**: Carpeta `data/predictions/`.

Mecanismo de Coordinación

El sistema utiliza un mecanismo de **Heartbeat y Locking Optimista** para coordinar el trabajo de los Workers sin un servidor central de bloqueos.

Campo `health`

En `models.csv`, cada modelo tiene un campo `health` que es un timestamp ISO. - **Bloqueo**: Cuando un Worker toma un modelo para entrenar, el DataBase actualiza `health` al tiempo actual. - **Heartbeat**: El Worker envía periódicamente señales que actualizan este timestamp.

Asignación de Tareas (`get_model_to_run`)

Cuando un Worker pide trabajo (`/models/torun`), el DataBase busca en `models.csv`: 1. Modelos con `status != completed`. 2. Que cumplan una de dos condiciones: - Nunca han sido tocados (health inválido/vacío). - Su `health` es **mayor a 20 segundos** en el pasado (indicando que el Worker anterior murió o abandonó la tarea).

Si encuentra uno, actualiza su `health` inmediatamente (lo "reserva") y lo devuelve al Worker.

Tolerancia a Fallos

Fallo de Worker

Si un Worker muere mientras entrena: 1. Deja de enviar heartbeats. 2. El campo `health` del modelo en el DataBase envejece. 3. Después de 20 segundos, `find_model_to_run` considera el modelo como "abandonado" y se lo asigna a otro Worker.

Fallo de DataBase

Si un nodo DataBase cae: 1. **Descubrimiento**: Los otros nodos detectan la caída mediante `check_ip_alive`. 2. **Limpieza**: Se ejecuta

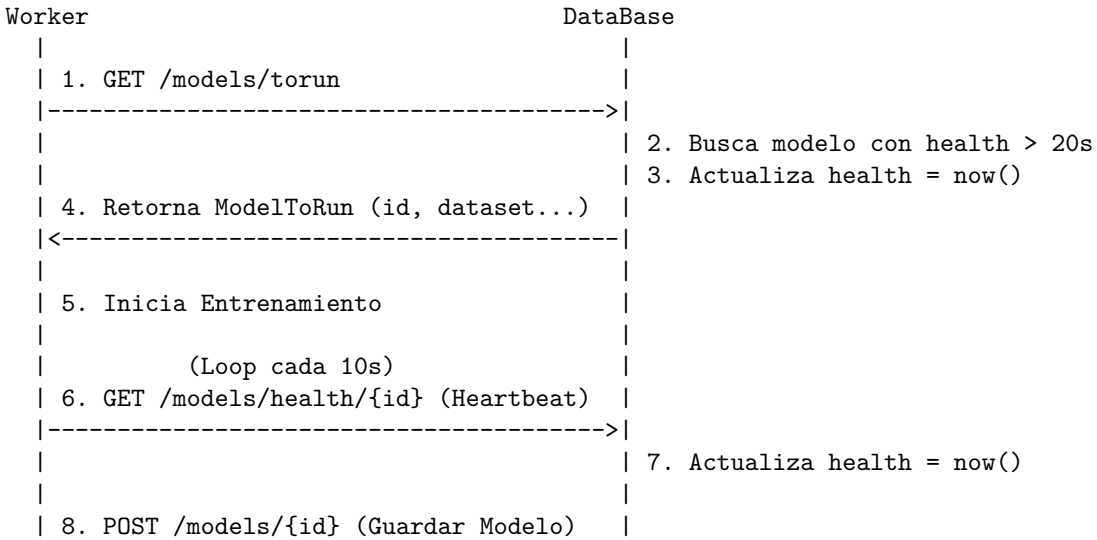
`cleanup_dead_peers`, eliminando al nodo caído de las listas de poseedores en `PeerMetadata`. 3. **Auto-Curación**: El `_replication_monitor` detecta que los datos que tenía ese nodo ahora tienen `replicas < REPLICATION_FACTOR`. 4. **Re-replicación**: Los nodos sobrevivientes que tienen copias de esos datos inician la replicación hacia otros nodos sanos para restaurar el factor de seguridad.

Consistencia Eventual

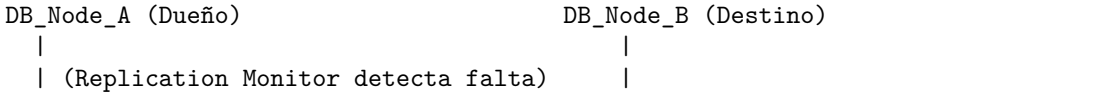
- **Merge de CSVs**: Al sincronizar metadatos, las filas de los CSVs se fusionan. Para `models.csv`, si hay conflicto, gana el estado `COMPLETED` o el timestamp `health` más reciente.
- **Comparación de Versiones**: Antes de servir un modelo (`GET /models/{id}`), el nodo verifica con sus pares si alguien tiene una "mejor" versión (ej. más entrenada o con más predicciones) y se actualiza a sí mismo si es necesario.

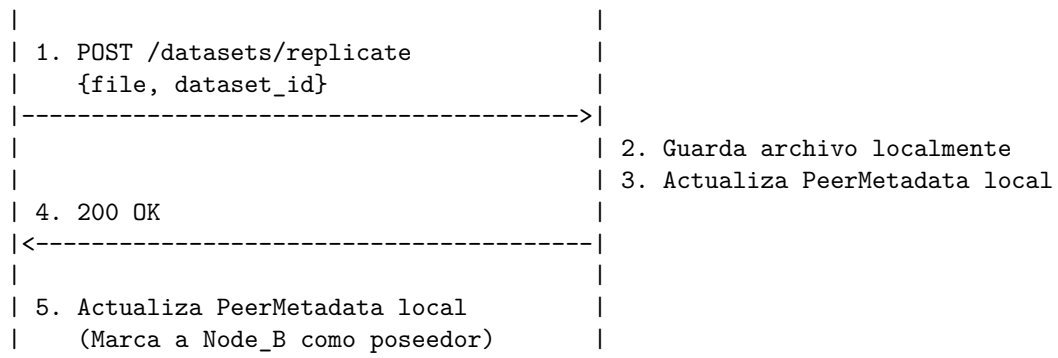
Diagramas

Flujo Worker DataBase (Entrenamiento)



Flujo DataBase DataBase (Replicación)





Comunicación y Protocolos

La arquitectura del sistema se basa en un modelo de comunicación **estrictamente HTTP/REST**, eliminando la necesidad de conexiones persistentes o sockets con estado entre los componentes (Workers, DataBases y Clientes).

Características Principales:

- **Desacoplamiento Total**: Cada nodo opera de forma autónoma. La caída o reinicio de un componente no afecta las conexiones de los demás.
- **API REST**: Tanto Workers como DataBases exponen interfaces RESTful estandarizadas para el intercambio de datos y comandos de control.
- **Escalabilidad**: Al no mantener estado en la capa de transporte, es posible agregar o quitar nodos dinámicamente sin reconfigurar la red.

Descubrimiento y Localización de Servicios

Para garantizar la alta disponibilidad y la resiliencia ante fallos de red, el sistema implementa una estrategia de descubrimiento híbrida:

1. Resolución DNS (Mecanismo Primario)

El sistema aprovecha el **DNS interno de Docker Swarm**. - **Resolución Automática**: Los nodos resuelven los nombres de servicio (ej. `dml-db`) para obtener las IPs de las instancias activas. - **Balanceo de Carga**: Docker distribuye las peticiones entre los contenedores disponibles. - **Aislamiento**: Permite que los nodos desconozcan la topología física de la red.

2. Caché de IPs (Mecanismo de Respaldo)

Para mitigar posibles fallos o latencias en el servicio DNS, cada nodo mantiene una tabla de enrutamiento local (`ip_cache`) gestionada por el Middleware.

Ciclo de Vida de la Caché: 1. **Inicialización**: Carga IPs semilla preconfiguradas para garantizar conectividad inmediata al arranque. 2. **Aprendizaje**

Continuo: Al descubrir un nuevo nodo sano (vía DNS o interacción directa), su IP se añade a la caché. 3. **Failover**: Si la resolución DNS falla, el sistema conmuta automáticamente al uso de IPs en caché. 4. **Auto-limpieza**: Un proceso de *Health Check* elimina proactivamente las IPs que dejan de responder, manteniendo la lista de pares depurada.

Topología y Distribución

Esta arquitectura distribuida permite que el sistema opere de manera funcional con una topología mínima de **un nodo Worker y un nodo DataBase**, escalando horizontalmente según la demanda y garantizando la continuidad del servicio incluso en escenarios de degradación parcial de la red.

Coordinación

El sistema implementa mecanismos de coordinación descentralizada para garantizar la consistencia y evitar condiciones de carrera sin depender de un coordinador central único.

Sincronización de Acciones

- **Workers**: Utilizan un mecanismo de **Heartbeat** para coordinar la asignación de tareas. No se comunican entre sí, sino que utilizan el estado compartido en los DataBases como medio de sincronización indirecta.
- **DataBases**: Se coordinan mediante el intercambio periódico de metadatos (`PeerMetadata`). Cada nodo informa a sus pares sobre los datos que posee, permitiendo que la red converja hacia un estado consistente.

Acceso Exclusivo a Recursos (Locking Optimista)

Para evitar que múltiples Workers entrenen el mismo modelo simultáneamente, se utiliza un sistema de bloqueo basado en timestamps (`health`): 1. **Adquisición**: Un Worker solicita un modelo (`get_model_to_run`). El DataBase verifica si el modelo está libre (health $>$ 20s) y actualiza el timestamp al momento actual, "reservándolo" efectivamente. 2. **Mantenimiento**: El Worker envía heartbeats cada 10s para renovar su reserva. 3. **Liberación**: Al finalizar, el estado cambia a `COMPLETED`. Si el Worker falla, el timestamp envejece y el recurso se libera automáticamente tras 20s (timeout).

Toma de Decisiones Distribuidas

- **Replicación**: La decisión de quién replica un dato faltante se toma de forma determinista y descentralizada. Solo el nodo poseedor con la **IP más baja** (menor orden lexicográfico) inicia la replicación, evitando duplicidad de esfuerzos.

- **Consistencia**: En caso de conflicto de versiones de metadatos, todos los nodos aplican la misma regla de resolución (prioridad a estado `COMPLETED` o timestamp más reciente), garantizando convergencia sin votación explícita.

Consistencia y Replicación

El sistema prioriza la disponibilidad y la tolerancia a particiones (AP en el teorema CAP), garantizando consistencia eventual.

Distribución de Datos

Los datos se distribuyen en la red de DataBases. No todos los nodos tienen todos los datos, pero el sistema garantiza que existan suficientes copias para asegurar la disponibilidad. - **Datasets**: Se almacenan particionados en archivos batch. - **Modelos**: Se almacenan como archivos JSON serializados. - **Predicciones**: Se guardan en archivos JSON asociados a modelos y datasets.

Replicación

- **Factor de Replicación**: El sistema mantiene por defecto **3 réplicas** (`REPLICATION_FACTOR = 3`) de cada dataset, modelo y predicción.
- **Mecanismo**: Un hilo de monitoreo (`_replication_monitor`) verifica periódicamente si los datos cumplen con el factor de replicación. Si detecta déficit, inicia la copia hacia nodos disponibles.

Confiabilidad tras Actualización

- **Metadatos**: La propagación de metadatos es epidémica (gossip-like). Un cambio en un nodo se propaga a los demás en los siguientes ciclos de sincronización (cada 7-10s).
- **Datos Voluminosos**: La replicación de archivos es asíncrona. Durante el breve periodo de propagación, es posible que algunos nodos no tengan la última versión, pero el mecanismo de descubrimiento de "mejor versión" mitiga el impacto de leer datos obsoletos.

Tolerancia a Fallas

El sistema está diseñado para operar en entornos inestables, asumiendo que los fallos son la norma y no la excepción.

Nivel de Tolerancia

- **Workers**: Tolerancia a fallos total mientras quede al menos 1 Worker activo ($N - 1$).

- **DataBases**: Tolerancia a la caída de $R - 1$ nodos por dato, donde R es el factor de replicación (por defecto 3). El sistema sigue operativo globalmente mientras exista al menos 1 nodo DataBase, aunque algunos datos específicos podrían no estar disponibles si se pierden todas sus réplicas.

Respuesta a Errores

- **Timeouts**: Todas las comunicaciones internas tienen timeouts configurados para evitar bloqueos indefinidos.
- **Reintentos**: Los clientes y nodos reintentan operaciones fallidas contra otros nodos disponibles.
- **Auto-Curación**: El sistema detecta nodos caídos y re-replica automáticamente los datos perdidos para recuperar el nivel de redundancia.

Fallos Parciales y Recuperación

- **Nodos Caídos Temporalmente**: Si un nodo cae y vuelve, recupera su estado mediante la sincronización de metadatos con sus pares.
- **Nodos Nuevos**: Se integran automáticamente al ser descubiertos por el DNS/Gossip, recibiendo metadatos y participando eventualmente en el almacenamiento de nuevas réplicas.

Seguridad

Seguridad del Diseño

- **Validación de Datos**: Todos los endpoints utilizan esquemas Pydantic estrictos para validar la estructura y tipos de datos de entrada, previniendo inyección de datos malformados.