

Diseño e implementación de una arquitectura de software crítico tolerante a fallos: Arquitectura Híbrida (Sentinel/Cluster) y diversidad de diseño para la detección de anomalías en entornos IoT industriales.

Diego Huerta Nuñez

diego.huertan@estudiantes.uv.cl
Aprendizaje Automático. Universidad de Málaga.

Abstract. El presente estudio aborda el desafío de implementar un sistema de detección de anomalías distribuido, resiliente y escalable para entornos SCADA/IoT en el contexto de la Industria 4.0. Ante el volumen masivo de datos generados por sensores y la incapacidad de los sistemas tradicionales para detectar anomalías sutiles, se diseñó un sistema de software crítico desplegado en la nube (GCP).

La metodología arquitectónica se centró en la flexibilidad ante el Teorema CAP, implementando una ****Arquitectura Dual**** configurable. El sistema permite operar bajo dos paradigmas: 1) ****Alta Disponibilidad con Consistencia Fuerte (CP)**** mediante Redis Sentinel, ideal para entornos donde la integridad del dato es prioritaria; y 2) ****Escalabilidad Horizontal (AP)**** mediante Redis Cluster con *Sharding* automático, para entornos de ingesta masiva.

Para garantizar la fiabilidad (R) de la detección, se empleó la técnica de Diversidad de Diseño, combinando cuatro modelos de Inteligencia Artificial (Determinista, Isolation Forest, Autoencoder y LSTM) bajo una regla de Voto por Consenso M-of-N (3 de 4). Esta estrategia reduce drásticamente la probabilidad de fallos comunes y minimiza las falsas alarmas.

El proyecto resolvió desafíos complejos de Ingeniería de Despliegue, incluyendo la gestión de dependencias (Dependency Hell) y la incompatibilidad binaria (Error SIGILL), forzando la compilación cruzada (linux/amd64) para garantizar la inmutabilidad. El resultado es un sistema validado con pruebas forenses de *failover* y *sharding*, con trazabilidad completa mediante Grafana.

1 Introducción: Visión Global y Objetivos Estratégicos

1.1 Definición del Problema: El Desafío de la Disponibilidad y Detección en la Industria 4.0

El contexto de la Industria 4.0 exige sistemas SCADA distribuidos capaces de manejar volúmenes masivos de datos IoT en tiempo real. Los sistemas tradicionales fallan al garantizar simultáneamente la escalabilidad horizontal y la detección de anomalías con alta fiabilidad. La infraestructura desplegada en la nube debe resolver el dilema constante del Teorema CAP (Consistencia, Disponibilidad, Tolerancia a Particiones), donde los fallos de red no son una excepción, sino una norma operativa.

1.2 Objetivo Principal: Desarrollo de un Sistema Distribuido Agnóstico a la Infraestructura

El objetivo fue diseñar y desplegar en IaaS (Google Cloud Platform) un sistema agnóstico a la infraestructura de datos. El sistema materializa una ****Arquitectura Dual (ADR 02)****, capaz de operar en modo Sentinel (CP) o Cluster (AP), utilizando un motor de IA robusto basado en consenso para minimizar falsos positivos y garantizar la integridad de la decisión.

1.3 Objetivos Específicos Cumplidos

Los objetivos se centraron en garantizar propiedades de software crítico:

1. **Arquitectura Dual:** Implementación flexible que soporta CP y AP mediante configuración, abordando el compromiso de diseño de sistemas distribuidos.
2. **Orquestación Distribuida:** Uso de Docker Swarm en GCP para demostrar conceptos de Alta Disponibilidad (HA) y *Sharding*.
3. **Inteligencia Robusta:** Motor de IA con Diversidad de Diseño y votación M=3 de 4 para mitigar fallos comunes y alucinaciones de modelos individuales.

2 Marco teórico y registro de decisiones de arquitectura (ADRs)

2.1 Justificación del sistema como crítico (Safety vs. Mission Critical)

El sistema desarrollado se clasifica como Software Crítico debido a su función central de detección de anomalías en entornos SCADA/IoT. Aunque el proyecto no opera directamente en el dominio *Safety Critical* (riesgo directo a la vida o al medio ambiente), se sitúa firmemente en el dominio *Mission Critical*, donde la pérdida de funcionalidad o la integridad del dato es inaceptable.

1. **Prioridad de la Fiabilidad (R) y Consistencia (C):** La criticidad deriva de la necesidad de garantizar la integridad de las series temporales y la fiabilidad (R) de la detección de anomalías. El diseño de la Arquitectura Dual (ADR 02) prioriza la Consistencia Estricta (CP) en el modo Sentinel para entornos donde la pérdida de datos o la violación de la secuencia es inaceptable.
2. **Mitigación de Fallos Correlacionados:** La criticidad se aborda mediante la ingeniería de la confiabilidad, específicamente en el motor de IA. El diseño se enfoca en mitigar el riesgo de Fallo de Causa Común (CCF), garantizando que la decisión de emitir una alerta (la misión crítica del sistema) no dependa de un único algoritmo.
3. **Resiliencia Distribuida como Requisito:** El despliegue en la nube (GCP) asume la Tolerancia a Particiones (P) como obligatoria. La capacidad probada de Auto-Curación (*Self-Healing*) ante la inyección de fallos catastróficos (Evidencias A y B) demuestra que el sistema cumple con el requisito *Mission Critical* de mantener la disponibilidad operativa y la integridad de la información bajo estrés.

2.2 ADR 01: Selección del orquestador (Docker Swarm)

Decisión: Docker Swarm.

Justificación: El orquestador fue seleccionado para demostrar de forma eficiente y controlada los conceptos fundamentales de los sistemas distribuidos que sustentan la Arquitectura Dual.

1. **Baja Sobrecarga para Demostración Académica:** Se seleccionó Docker Swarm por su menor sobrecarga operativa en comparación con Kubernetes, lo cual era ideal para demostrar los conceptos de Alta Disponibilidad (HA) y *Sharding* en un entorno académico acotado.
2. **Gestión de Topologías HA y Sharding:** Swarm facilita la gestión de servicios replicados y el despliegue de redes *overlay* cifradas. Esto permitió la implementación efectiva de los requisitos de topología del ADR 02: desplegando la arquitectura Sentinel (HA) y la arquitectura Cluster (*Sharding* de 16384 slots).
3. **Resiliencia de Infraestructura y Mitigación de SPOF:** Swarm, en su topología de 3 nodos (1 Manager, 2 Workers), valida la capacidad de orquestación de la resiliencia. Específicamente, las restricciones de ubicación (*placement constraints*) fueron clave para prevenir el Fallo de Causa Común (CCF) al asegurar que el Maestro y la Réplica de Redis nunca coexistan en el mismo nodo físico.

La arquitectura resolvió el compromiso del Teorema CAP mediante la implementación de una Arquitectura Dual configurable. La **Tolerancia a Particiones (P)** se consideró obligatoria al desplegarse en la nube, forzando la elección constante entre Consistencia (C) y Disponibilidad (A).

Modo Operativo	Decisión Teórica	Topología y Justificación
Modo A (Sentinel)	Prioridad CP (Consistencia Fuerte)	Topología 1 Maestro, 1 Réplica, 3 Sentinels. Ideal para entornos críticos, garantiza un único punto de escritura activo para la integridad del dato.
Modo B (Cluster)	Prioridad AP (Disponibilidad y Particionamiento)	Topología 6 Nodos con <i>Sharding</i> de 16384 slots. Ideal para ingesta masiva donde la escalabilidad es prioritaria.

Table 1. Estrategia Dual del ADR 02

2.3 ADR 03: Estrategia de robustez de IA mediante Diversidad de Diseño y Consenso M-of-N

Decisión: Voto por Consenso M-of-N (3 de 4) con Diversidad de Diseño.

Justificación: El diseño del subsistema de IA se basa en los principios de ingeniería de software crítico para mitigar la incertidumbre y el riesgo inherente de los modelos de Machine Learning.

1. **Ortogonalidad de los Modos de Fallo (Diversidad de Diseño):** Para aumentar la fiabilidad y mitigar el riesgo de Fallo de Causa Común (CCF), se empleó una Diversidad Algorítmica Fundamental. Los cuatro modelos implementados son epistemológicamente disjuntos: la Regla Física (determinista), Isolation Forest (estadístico), Autoencoder (reconstrucción) y LSTM (secuencial).
2. **Lógica de Quórum (M=3):** Se exige un consenso de $M = 3$ votos para declarar una anomalía como crítica. Este umbral es una decisión de ingeniería crítica que reduce drásticamente los Falsos Positivos (FP) causados por el ruido o las alucinaciones de modelos individuales, sin incurrir en Falsos Negativos (FN) si uno de los modelos (ej. el LSTM) falla operativamente.
3. **Garantía de Consistencia Predictiva:** Para asegurar que la Replicación de Máquina de Estados (SMR) y el consenso sean deterministas, se adoptó la Inmutabilidad (ADR 04). Los modelos de IA se entrenan *offline* y se empaquetan de forma inmutable en la imagen Docker (*Build-Time Injection*), garantizando que las 5 réplicas del servicio API usen exactamente la misma lógica predictiva y no diverjan en la votación.

3 Arquitectura técnica detallada y estrategia de despliegue

3.1 Infraestructura IaaS en Google Cloud Platform (Topología y Red Overlay)

El sistema se desplegó en Google Cloud Platform (GCP) bajo un modelo IaaS, utilizando máquinas virtuales (VMs) de Compute Engine para simular un clúster de producción con requisitos de alta disponibilidad y gestión distribuida.

1. **Topología Física y Orquestación (ADR 01):** El despliegue se realizó sobre un clúster Docker Swarm de 3 nodos (3 VMs e2-medium en `us-central1`): 1 Manager y 2 Workers. Swarm fue elegido por su menor sobrecarga operativa, ideal para demostrar los conceptos de HA y *Sharding* en un entorno controlado.
2. **Red Overlay Cifrada:** Toda la comunicación entre los microservicios, incluyendo las 5 réplicas de la API y los nodos de Redis, se realiza a través de una Red Overlay (Virtual Private Cloud - VPC) configurada y cifrada. Esto garantiza el aislamiento del tráfico Swarm (puertos 2377/7946/4789) y protege las comunicaciones internas de la capa de datos.
3. **Mitigación de SPOF Físico:** La arquitectura mitiga el riesgo de SPOF (*Single Point of Failure*) y el Fallo de Causa Común (CCF) físico. Se utilizaron Restricciones de Ubicación (*placement constraints*) en el `docker-stack.yml` para asegurar que el nodo Maestro de Redis y su Réplica (en modo Sentinel) nunca coexistan en el mismo nodo físico, garantizando la resiliencia en caso de fallo de una VM.

3.2 Componentes de software y stack tecnológico (Microservicios)

La arquitectura se organiza en tres capas fundamentales (Servicio, Datos, Observabilidad), utilizando tecnologías de código abierto maduras para asegurar la estabilidad y el desempeño en el manejo de series temporales.

1. **Capa de Servicio (API Gateway):** Implementada con FastAPI y servida por Gunicorn. Esta capa aloja la lógica de negocio y el motor de IA. Se despliega con 5 réplicas para asegurar la Alta Disponibilidad (HA) y gestionar la concurrencia de peticiones. Su configuración es agnóstica a la capa de datos gracias al Patrón Factory.
2. **Capa de Datos (Dual-Mode):** Utiliza Redis Stack Server para ofrecer soporte nativo a RedisTimeSeries en ambas configuraciones arquitectónicas (Sentinel y Cluster). Esta elección tecnológica es crucial, ya que permite la inserción y consulta de rangos temporales con complejidad algorítmica $O(1)$, un requisito crítico para la baja latencia del dashboard.
3. **Capa de Observabilidad:** Compuesta por Grafana. Grafana consume directamente los datos de Redis a través del módulo TimeSeries, actuando como la caja negra del sistema para el análisis forense y la trazabilidad de la decisión de IA.

3.3 Servicio web (FastAPI): Patrones de diseño para la dualidad

La API Gateway (FastAPI) es la capa de software que materializa el **ADR 02 (Arquitectura Dual)**, separando la lógica de negocio de la complejidad de la infraestructura de datos mediante patrones de diseño.

1. **Patrón de Diseño Factory (Switch de Infraestructura):** La conmutación entre el Modo CP (Sentinel) y el Modo AP (Cluster) se implementa mediante el Patrón Factory en el módulo `database.py`. Este patrón lee la variable de entorno `REDIS_MODE` definida en el stack de orquestación, instanciando el cliente de conexión (`RedisCluster` o `SentinelConnection`) de manera condicional.
2. **Principio Abierto/Cerrado (Coherencia Teórica):** Este diseño garantiza que el *Service Layer* de la API sea cerrado a la modificación al cambiar la topología de Redis. La lógica de negocio solo interactúa con una interfaz de repositorio abstracta, sin necesidad de recompilación de código, cumpliendo con los principios de arquitectura limpia (*Clean Architecture* - ADR 04).
3. **Patrón de Resiliencia (Retry):** Se implementó el patrón **Retry con Exponential Backoff**. Este mecanismo es crítico para la resiliencia en el Modo Sentinel (CP), ya que permite a la API esperar el RTO (< 10 s) durante el *failover* sin fallar catastróficamente, para luego reanudar las escrituras y garantizar la Ordenación Total de las Series Temporales.

3.4 Estrategia de Persistencia Híbrida: Abordando el Compromiso CAP

Modo Consistencia Estricta (CP): Topología Sentinel y Failover Activo/Pasivo El Modo Consistencia Estricta se define como la Opción A dentro del **Registro de Decisiones de Arquitectura ADR 02: Estrategia de Base de Datos**, y está diseñado para aquellos escenarios SCADA donde la integridad del dato es prioritaria y la pérdida de información es inaceptable.

Justificación Teórica: Prioridad CP En línea con el Teorema CAP, esta topología prioriza la Consistencia (C) y la Tolerancia a Particiones (P), inevitablemente sacrificando temporalmente la Disponibilidad (A) durante el proceso de conmutación. La elección **CP** se justifica porque garantiza un **único punto de escritura** activo en todo momento. Esta unicidad es crítica para mantener la Ordenación Total de las series temporales, evitando que múltiples nodos acepten datos contradictorios en caso de partición de red.

Topología y Failover Activo/Pasivo La arquitectura se configura con un patrón activo/pasivo simple, monitorizado por un sistema de orquestación externo:

- **Topología:** 1 Maestro, 1 Réplica y 3 Nodos Sentinel.
- **Función de Sentinel:** Los 3 Sentinels actúan como un **Quórum de Consenso** que monitorea constantemente el estado de los nodos Redis. Si el Maestro deja de responder, el quórum de Sentinels debe alcanzar una mayoría (al menos 2 de 3) para declarar el fallo mediante el estado *SDown* (Subjectively Down) y luego *ODown* (Objectively Down), procediendo a la elección de un nuevo Maestro.

Mitigación del Split-Brain mediante Coordinación La mitigación del escenario de “Split-Brain” (donde el Master antiguo podría seguir aceptando escrituras tras la partición, creando inconsistencia) se garantiza en dos niveles:

1. **Principio de Quórum:** El Sentinel, al promover a la Réplica, ejecuta comandos para degradar al antiguo Maestro y reconfigurar la Réplica, adhiriéndose a un principio de coordinación distribuida (similar a la Sincronía Virtual) que exige una “vista” acordada por la mayoría de los Sentinels.
2. **Enforcement de la Aplicación:** El diseño de la API Gateway implementa un mecanismo defensivo que garantiza la Consistencia Estricta (C). Durante la fase de elección del nuevo líder (el periodo de indisponibilidad temporal), **la API rechaza las peticiones de escritura**. Esto previene activamente que el sistema acepte datos que podrían caer en un nodo obsoleto o inaccesible, asegurando que la única verdad (el nuevo Master) sea el único punto de aceptación de datos.

Métrica de Recuperación Crítica (RTO) La métrica fundamental para evaluar la robustez en este modo es el **Objetivo de Tiempo de Recuperación (RTO)**, que mide la latencia introducida por el *failover*. En el contexto de SCADA, el RTO debe ser mínimo para evitar la pérdida prolongada de muestras. La **Evidencia Forense A** (Validación, Capítulo 6) validó que, tras la simulación de un fallo catastrófico (`docker kill`), el sistema fue capaz de detectar el fallo, promover la réplica y reconfigurar la conexión para la API en menos de 10 segundos (< 10 s). Este resultado cumple con los requisitos de Alta Disponibilidad para entornos críticos no-life-support.

Modo Alta Disponibilidad (Sentinel/CP): Prioridad a la Consistencia ...

1. **Defensa Teórica y Resultado FLP:** El Mecanismo de Bloqueo de Escritura de la API durante el *failover* (la ventana crítica de Partición) no es solo una mitigación del *Split-Brain*, sino una **respuesta de ingeniería al límite teórico impuesto por el Resultado de Imposibilidad FLP** (Fisher, Lynch, Paterson). Al sacrificar explícitamente la Disponibilidad (A) ante el fallo, se garantiza que el sistema no intente un consenso atómico inviable en un sistema distribuido asíncrono, preservando la **Consistencia (C)** de forma incondicional.

Modo Escalabilidad y Particionamiento (AP): Topología Redis Cluster y Sharding La implementación del modo Cluster se basa en el **Registro de Decisiones de Arquitectura ADR 02**, que define la necesidad de una capacidad de **Escalabilidad Horizontal (AP)** para manejar escenarios de ingesta masiva de datos SCADA sin que un único nodo se convierta en un cuello de botella.

Justificación Teórica y Compromiso CAP De acuerdo con el Teorema CAP, en presencia de una partición de red (*P*), la elección debe recaer entre Consistencia (*C*) o Disponibilidad (*A*). Para el modo Cluster, la arquitectura prioriza *A* y *P*. Al elegir *AP*, se garantiza que el sistema continúe aceptando escrituras y entregando lecturas incluso si una parte del *cluster* se aísla de la red. La implicación práctica de este compromiso es la relajación de la Consistencia (*C*). Sin embargo, esta elección se justifica bajo las siguientes premisas de diseño en el contexto de series temporales:

- **Tolerancia a Particiones como Obligatoria:** Dado que el sistema se despliega en IaaS (Google Cloud Platform) utilizando Docker Swarm, las particiones de red son un hecho inevitable en entornos distribuidos. La tolerancia a particiones es, por lo tanto, una garantía obligatoria.

- **Series Temporales Monotónicas:** Los datos SCADA (ej. `redis-timeseries`) son inherentemente monotónicos y secuenciales. La necesidad crítica no es la Consistencia Global Inmediata, sino la **Disponibilidad para la Ingesta**, con la garantía de que el dato eventualmente alcanzará la consistencia.

Topología y Mecanismo de Sharding La topología del modo Cluster se configura mediante 6 nodos distribuidos: 3 Maestros y 3 Réplicas. Este despliegue aprovecha el particionamiento nativo de Redis Cluster, distribuyendo el espacio de claves de manera determinista:

- **Distribución de Slots:** El espacio de claves se fragmenta en **16384 slots**. Cada nodo Maestro es responsable de una porción de estos slots, y el cliente Redis (el driver `redis-py` utilizado por la API) se encarga de determinar qué nodo aloja una clave específica antes de enviar la petición.
- **Manejo de Fallos:** Si un nodo Maestro falla, su Réplica asciende automáticamente (*Failover*). El cliente API gestiona la redirección de solicitudes a través del protocolo nativo del cluster, manejando internamente los errores `MOVED` de forma transparente a la capa de servicio. Esta resiliencia se valida empíricamente en la **Evidencia Forense B** (Capítulo 6).

Garantía de Ordenación Total para Series Temporales Un requisito fundamental de la Replicación de Máquina de Estados (SMR) y la gestión de series temporales es la **Ordenación Total de Mensajes** para que todas las réplicas procesen comandos en el mismo orden exacto. Para garantizar que el *sharding* no comprometa la integridad temporal de una misma serie, se utiliza la regla de *hashing* de Redis Cluster:

- **Localidad de Claves:** Una clave única (ej. `sensor:temperatura:1`) se asigna de forma determinista a un **único slot** y, por extensión, a un único nodo Maestro.
- **Garantía FIFO:** Dado que todas las escrituras para esa clave se dirigen al mismo punto de escritura, se asegura la ordenación estricta FIFO (*First-In, First-Out*) de los comandos emitidos por el cliente para esa serie.
- **Redis TimeSeries:** La elección de RedisTimeSeries (disponible en `redis-stack-server`) refuerza esta garantía, ya que está optimizado para la inserción y consulta de rangos temporales $O(1)$ dentro del contexto de un único dataset, manteniendo la integridad secuencial necesaria para el análisis posterior por los modelos de IA.

3.5 Capa de observabilidad y dashboard (Grafana)

La observabilidad no es considerada una conveniencia operativa, sino una **condición crítica para el Aseguramiento (Assurance)** del sistema. Esta capa valida que la información generada por la Arquitectura Dual y el motor de IA por Consenso sea auditable y trazable en tiempo real.

1. **Rol de Caja Negra y Evidencia Forense (Safety Case):** El dashboard de Grafana actúa como la **caja negra** del sistema. Esto es esencial para el **Estudio de la Seguridad (Safety Case)**, ya que permite la **Reconstrucción Histórica** de la secuencia exacta de eventos. Es la interfaz que verifica el cumplimiento de las métricas de ingeniería crítica, como la validación empírica de que el Objetivo de Tiempo de Recuperación (RTO) fue de < 10 segundos tras un *failover* (Evidencia Forense A).
2. **Explicabilidad (XAI) y Consenso Determinista:** La observabilidad extiende la confiabilidad del motor de IA al exponer los datos de la lógica de consenso (**ADR 03**). El dashboard no solo muestra el resultado binario final (`es_anomalia: true`), sino también el **desglose de los 4 votos individuales** (Regla Física, Isolation Forest, Autoencoder, LSTM). Esto provee **trazabilidad de la decisión**, manteniendo al operador humano en el bucle (*Human-in-the-loop*) para validar la coherencia y mitigar Falsos Positivos Parciales.
3. **Eficiencia Algorítmica $O(1)$ en la Monitorización Crítica:** La capacidad de visualización en tiempo real (un requisito no funcional crítico para SCADA) se sustenta en la elección de la tecnología de persistencia. El uso de **RedisTimeSeries** garantiza que la inserción y la recuperación de rangos temporales para el dashboard se realicen con una complejidad algorítmica tendiente a $O(1)$. Esta eficiencia es vital para asegurar que la monitorización no sufra latencia (*lag*) incluso con un histórico masivo de datos.

4 El subsistema de inteligencia artificial: diseño de robustez mediante diversidad y consenso

4.1 Justificación de la diversidad de modelos (Reglas Físicas, Estadísticos y Deep Learning)

El diseño del subsistema de IA se basa en la técnica de Diversidad de Diseño (ADR 03) para lograr la Ortogonalidad de los Modos de Fallo. Esta estrategia es crucial en software crítico porque mitiga el riesgo de Fallo de Causa Común (CCF), donde la ambigüedad en la especificación haría que múltiples modelos fallaran ante la misma entrada.

1. **Ortogonalidad Epistemológica para Mitigar CCF:** Los cuatro modelos implementados son paradigmas matemáticos y de Machine Learning fundamentalmente disjuntos, lo que garantiza que la debilidad de un enfoque no se propague a los demás. La diversidad incluye: Regla Física (determinista), Isolation Forest (estadístico), Autoencoder (reconstrucción, Deep Learning) y LSTM (secuencial, Deep Learning).
2. **Abordaje de Riesgos Múltiples:** La combinación asegura una cobertura amplia contra diferentes tipos de anomalías. El Autoencoder detecta fallos estructurales por error de reconstrucción, mientras que el LSTM detecta desviaciones secuenciales complejas no evidentes en un único punto de dato. La Regla Física y el Isolation Forest actúan como filtros robustos y de bajo costo computacional.
3. **Garantía de SMR mediante Inmutabilidad:** La base de la diversidad debe ser determinista. La Inmutabilidad (ADR 04) garantiza que todos los contenedores de la API (5 réplicas) utilicen exactamente los mismos modelos entrenados. Al “quemar” los modelos en la imagen Docker durante el *Build-Time*, se asegura que no haya divergencia de pesos o lógica predictiva entre nodos, un requisito esencial para la Replicación de Máquina de Estados (SMR) y la coherencia del consenso.

4.2 El proceso de agregación y la lógica de Voto por Consenso (M-of-N)

La lógica de consenso es la capa final de protección, transformando las salidas probabilísticas e inciertas de los modelos de IA en una decisión determinista y confiable para el entorno SCADA.

1. **Mecanismo de Votación Paralela:** El servicio *AnomalyService* ejecuta los cuatro modelos de forma paralela. Cada modelo emite un voto binario (0 o 1), indicando si la entrada constituye una anomalía. Este enfoque paralelo acelera la toma de decisiones y mantiene la latencia baja.
2. **Umbral de Consenso $M=3$ para Optimización de Riesgos:** La regla exige un consenso de $M = 3$ de 4 votos para emitir una alerta crítica. Esta elección técnica resuelve el compromiso entre Seguridad y Disponibilidad Operativa. Exigir $M = 3$ mitiga Falsos Positivos (FP) (ruido o alucinaciones de un solo modelo) sin ser tan estricto como $M = 4$ (unanimidad), lo que previene Falsos Negativos (FN) si un modelo complejo falla silenciosamente.
3. **Integridad Secuencial (Serialización Atómica):** El proceso de consenso no puede operar sobre datos inconsistentes. Aunque la API opera con 5 réplicas concurrentes, la Ordenación Total de Mensajes se garantiza delegando la serialización al Maestro de Redis. Redis opera con un *Single-Threaded Event Loop*, lo que lo convierte en el Serializador Atómico del sistema. Los modelos secuenciales (como el LSTM) siempre consultan la fuente de verdad serializada en *RedisTimeSeries*, eliminando las condiciones de carrera en la evaluación del consenso.

5 Desafíos de ingeniería y soluciones implementadas: Crónica del desarrollo crítico

5.1 El Teorema CAP como Constante de Diseño: La Obligatoriedad de P

La ingeniería de la capa de datos comenzó con la aceptación de que la **Tolerancia a Particiones (P)** es una condición obligatoria (*mandatory*) en cualquier despliegue en la nube (GCP/Swarm), ya que las redes son asíncronas y no fiables por definición.

Justificación Técnica: Al ser P una constante física del entorno, el Teorema CAP dicta que cuando la red falla, el sistema debe elegir matemáticamente entre:

1. Bloquearse para mantener la verdad (Consistencia - CP, modo Sentinel).
2. Seguir operando con datos potencialmente divergentes (Disponibilidad - AP, modo Cluster).

La Arquitectura Dual fue la respuesta a este dilema, ofreciendo una solución técnica y teórica para cada necesidad operativa.

5.2 El Error SIGILL y el Determinismo Requerido para SMR

El desarrollo del subsistema de IA enfrentó un desafío de compatibilidad binaria: el error **SIGILL** (*Illegal Instruction*). TensorFlow compilado con optimizaciones agresivas para AVX2 fallaba en las CPUs virtuales de GCP.

Relación con el Determinismo (SMR): La resolución de este error fue un requisito fundamental para la **Replicación de Máquina de Estados (SMR)**. SMR exige que todas las 5 réplicas de la API produzcan exactamente la misma salida ante la misma entrada. Las operaciones de punto flotante en Deep Learning pueden introducir sutiles variaciones entre diferentes arquitecturas de CPU.

Solución Técnica: Se forzó la compatibilidad binaria mediante la desactivación de optimizaciones (`TF_ENABLE_ONEDNN_OPTS=0`) y la fijación estricta de la versión (`tensorflow==2.15.0`). Esto eliminó el no-determinismo del hardware subyacente, asegurando que la inferencia matemática fuera idéntica en todas las réplicas, validando que el consenso $M = 3$ fuera real y no producto de una divergencia computacional.

5.3 Gestión de dependencias y el infierno de Python (Dependency Hell)

El desarrollo del stack de IA enfrentó el desafío clásico del “Infierno de Dependencias” (*Dependency Hell*), donde las bibliotecas complejas (TensorFlow, NumPy, Pandas) requerían versiones incompatibles entre sí, poniendo en riesgo la estabilidad y el determinismo del contenedor.

1. **Conflicto de Compatibilidad Binaria:** El problema principal residía en la matriz de compatibilidad de Deep Learning. Versiones antiguas de TensorFlow entraban en conflicto con las versiones de alto rendimiento de `numpy` y `pandas`. Esto obligó a tratar la matriz de versiones no como una lista flexible, sino como un **contrato de ingeniería de configuración**.
2. **Fijación Estricta de Versiones:** La solución fue establecer una matriz de compatibilidad estricta y moderna, reflejada en el `requirements.txt`. Se fijaron las versiones específicas (`numpy==1.26.4`, `pandas==2.2.0`, `tensorflow==2.15.0`). Esto resolvió los conflictos de versiones, garantizando la estabilidad y la reproducibilidad del `build` del contenedor.
3. **Garantía de SMR y Estabilidad:** Esta fijación estricta fue crítica para la **Replicación de Máquina de Estados (SMR)**. Al eliminar las fuentes de variabilidad en las dependencias, se complementó la solución del error SIGILL, asegurando que todas las 5 réplicas de la API utilizaran las mismas bibliotecas y, por ende, la misma lógica matemática determinista.

5.4 Inmutabilidad de los artefactos y el build-time vs. runtime

La estrategia de inmutabilidad (**ADR 04**) fue fundamental para asegurar la Consistencia Predictiva del motor de IA por consenso, especialmente ante la complejidad de la compilación cruzada (Mac/ARM64 vs. GCP/AMD64).

1. **Inyección de Artefactos (Build-Time Injection):** Para evitar la inconsistencia entre entornos (el riesgo de que un modelo entrenado en Mac ARM64 fallara al ejecutarse en Linux AMD64), se adoptó una estrategia de **Inyección en Tiempo de Construcción**. Los modelos (`.h5`, `.joblib`) se entrenaron *offline* y se integraron a la imagen Docker de forma inmutable mediante la directiva `COPY` en el `Dockerfile`.

2. **Garantía de Identidad Funcional:** Esta estrategia asegura que los modelos sean un componente inmutable del sistema de archivos del contenedor. Dado que todas las 5 réplicas se instancian a partir del mismo *Image Digest* SHA256, se garantiza que la divergencia en los pesos o parámetros del modelo es matemáticamente imposible.
3. **Consistencia Predictiva (ADR 03):** La inmutabilidad garantiza la Identidad Funcional necesaria para el motor de consenso. Si todas las réplicas tienen modelos idénticos, su votación será coherente, fortaleciendo la confiabilidad del quórum $M = 3$.

```
to register factory for plugin cuBLAS when one has already been registered
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Started server process [35]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Started server process [33]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Started server process [34]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Started server process [32]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Application startup complete.
```

Fig. 1. Carga de Modelos y Conexión Determinista. Logs de contenedores de la API (*industrial_web*) que verifican la conexión exitosa al Maestro de Redis después de cargar la lógica de infraestructura. La aparición de las marcas verdes “Conectado a Redis Master” en las 5 réplicas valida que el *Patrón Factory* y los modelos inyectados se inicializaron correctamente en todos los nodos, asegurando la **Consistencia Predictiva** (ADR 04).

```
to register factory for plugin cuBLAS when one has already been registered
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Started server process [35]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Started server process [33]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Started server process [34]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Started server process [32]
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Waiting for application startup.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:Conectando a Sentinel en sentinel:26379...
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [35] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [34] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [33] [INFO] Application startup complete.
industrial_web.5.9rmllyko2ndt@worker-01 | INFO:infrastructure:✓Conectado a Redis Master
industrial_web.5.9rmllyko2ndt@worker-01 | [2025-11-27 18:28:05 +0000] [32] [INFO] Application startup complete.
```

Fig. 2. Carga de Modelos y Conexión Determinista. Logs de contenedores de la API (*industrial_web*) que verifican la conexión exitosa al Maestro de Redis después de cargar la lógica de infraestructura. Múltiples réplicas (workers) muestran las marcas verdes de conexión, validando que la **Inyección de Artefactos en Build-Time** (ADR 04) y el *Patrón Factory* se inicializaron correctamente en todos los nodos, asegurando la **Consistencia Predictiva** (SMR).

5.5 Ingeniería de resiliencia: Patrones Retry, Exponential Backoff y Healthchecks

La ingeniería de resiliencia se aplicó para mitigar fallos en el arranque, la orquestación y, crucialmente, durante los eventos de *failover* de Redis Sentinel, asegurando la continuidad del servicio.

1. **Patrón Retry con Exponential Backoff:** Este patrón se implementó en el módulo de conexión a la base de datos (`database.py`) para resolver el problema de arranque (*start-up*). Permite a la API intentar la conexión (ej. 5 veces con esperas crecientes: 2s, 4s, 8s) si Redis aún no está disponible. Durante el *failover* de Sentinel, este mismo patrón es el que garantiza la **Ordenación Total de Mensajes** al retener la intención de escritura hasta que se establece la nueva conexión con el Maestro promovido.
2. **Healthchecks Personalizados y Self-Healing:** Se implementó un *Healthcheck* dedicado (`/health`) que verifica la conectividad de Redis. Si el *Healthcheck* falla, Docker Swarm utiliza esta información para aplicar políticas de **auto-curación (Self-Healing)**. Se configuró un *start-period* de 30 segundos para evitar que el *Healthcheck* fallara prematuramente durante el tiempo normal de inicialización de las dependencias.
3. **Mitigación de SPOF Lógico:** Los *Healthchecks* combinados con la replicación (5 réplicas de la API) aseguran que el orquestador detecte y reemplace rápidamente cualquier contenedor que pierda conexión con la capa de datos, manteniendo la disponibilidad operativa (A) de la capa de servicio.

5.6 Análisis de riesgos residuales (FMEA) en arquitectura distribuida

El diseño de la arquitectura dual y del motor de IA incluye una mitigación explícita de los riesgos críticos, lo que se asemeja a un **Análisis de Modos de Fallo y Efectos (FMEA)**, centrado en la protección de la integridad de los datos y la fiabilidad de las decisiones.

1. **Riesgo de Split-Brain (Sentinel):** El modo de fallo más grave en CP. **Mitigación:** La API implementa la **Prevención de Fallos (Fault Avoidance)** al rechazar intencionalmente las escrituras durante la ventana de *failover* (RTO). Esto garantiza la **Consistencia Estricta (C)** y evita que nodos obsoletos acepten datos, protegiendo la integridad de la serie temporal.
2. **Riesgo de Fallo de Causa Común (CCF) en IA:** La posibilidad de que todos los modelos fallen simultáneamente por un error lógico común. **Mitigación:** Se implementó la **Diversidad de Diseño (ADR 03)**, utilizando cuatro paradigmas matemáticos ortogonales (Físico, Estadístico, Deep Learning). Esto hace que sea “altamente improbable” que los cuatro fallen a la vez ante una anomalía real.
3. **Riesgo de Falsos Negativos (FN):** La posibilidad de no detectar una anomalía crítica. **Mitigación:** El quórum $M = 3$ garantiza que la alerta se emita incluso si uno de los modelos (ej., el LSTM) falla operacionalmente (un fallo silencioso), asegurando que la misión crítica del sistema (la detección) se cumpla ante la falla de un componente.

6 Validación, resultados empíricos y evidencia forense de funcionamiento

6.1 Verificación del despliegue en GCP y Topología Swarm

La fase inicial de validación se centró en la verificación del despliegue de la infraestructura como servicio (IaaS) para confirmar que la topología Swarm estaba correctamente configurada para soportar las arquitecturas duales (Sentinel y Cluster) y sus requisitos de resiliencia (**ADR 01**). Esta fase no solo demuestra la funcionalidad, sino que establece el **Estudio de la Seguridad (Safety Case)**, donde las mediciones empíricas se convierten en la **Evidencia Forense** necesaria para validar las aserciones de tolerancia a fallos.

1. **Topología IaaS y Red Overlay Cifrada:** El sistema se desplegó y verificó sobre 3 Máquinas Virtuales (VMs) (`e2-medium` en `us-central1`) de Google Cloud Platform (GCP), organizadas en un clúster **Docker Swarm** (1 Manager y 2 Workers). Se confirmó que la comunicación interna entre las 5 réplicas de la API y la capa de datos (Redis) se realizaba exclusivamente a través de una **Red Overlay cifrada**, aislando el tráfico Swarm (puertos 2377/7946/4789) y cumpliendo los requisitos de seguridad básica.
2. **Mitigación de SPOF Físico (Restricciones de Ubicación):** Se validó que el archivo `docker-stack.yml` incluyó las **Restricciones de Ubicación** (*placement constraints*). Este diseño fue esencial para la Arquitectura Dual, ya que garantiza que el nodo Maestro de Redis y su Réplica (en modo Sentinel) nunca coexistan en el mismo nodo físico (VM). Esto mitiga el riesgo de **Fallo de Causa Común (CCF)** a nivel de hardware, donde la pérdida de una VM completa paralizaría el *stack* de persistencia.

3. **Validación del Self-Healing de Orquestación:** El funcionamiento de Docker Swarm fue probado mediante la simulación de la pérdida catastrófica de un nodo de trabajo (*worker-1*). Se verificó que el orquestador detectó el estado Down del nodo y activó automáticamente el proceso de **Auto-Curación (Self-Healing)**. Swarm reprogramó las tareas (las réplicas perdidas de la API) en los nodos supervivientes (Manager o Worker-2), manteniendo el requisito de 5 réplicas y asegurando que la capa de servicio cumpliera con su objetivo de Disponibilidad.

6.2 Evidencia Forense A: Resiliencia y Recuperación de Fallos (Modo Sentinel/CP)

The screenshot displays Docker logs from a host named 'diaphos24manager'. The logs show the state of various services and containers. Key events include:

- Redis Master Failure:** The Redis master service on 'worker-02' transitions from 'Running' to 'Shutdown'.
- Sentinel Detection:** The Redis Sentinel service on 'worker-01' detects the master failure and initiates a failover process.
- Failover Process:** The logs show the Sentinel service performing a 'switch-master' operation, promoting a new master from the available slaves.
- Service Recovery:** The Redis master service is restarted on 'worker-02' and returns to a 'Running' state.

Fig. 3. Evidencia Forense A: Transición de Fallo en Sentinel. Los logs del Sentinel manager muestran el protocolo de consenso distribuido activándose tras el *docker kill* del Maestro. Se observa el paso de la detección subjetiva a la objetiva (+sdown → +odown) y el mensaje +switch-master, que confirma el *failover* automático. La ventana de tiempo entre el fallo y el *switch* valida el **RTO < 10 segundos**.

Esta prueba forense tuvo como objetivo la validación empírica de la Arquitectura Dual en su modo de **Alta Disponibilidad/Consistencia Fuerte (CP)**, tal como se definió en el ADR 02. La ejecución simula un fallo catastrófico e inesperado del nodo primario de la capa de datos.

Hipótesis de Prueba (H1) Si el nodo Maestro de Redis cae abruptamente, el quórum de Sentinels debe alcanzar el consenso, elegir y promover una nueva réplica a Maestro en un tiempo de recuperación objetivo (RTO) aceptable. La capa de servicio (API) debe reanudar las escrituras automáticamente, preservando la Ordenación Total de la serie temporal.

Procedimiento Experimental: El “Asesinato” del Maestro

1. **Configuración Inicial:** Despliegue del stack en modo Sentinel: 1 Maestro, 1 Réplica y 3 Nodos Sentinel.
2. **Inyección de Tráfico:** Se inicia un proceso concurrente de escritura de datos de series temporales (/nuevo).
3. **Inyección de Fallo:** Se ejecuta una orden de terminación forzada (*docker kill*) sobre el contenedor del nodo Maestro de Redis, simulando un fallo hardware o de sistema operativo.
4. **Medición y Observación:** Se monitorizan los logs de los Sentinels y el comportamiento de la API para medir el tiempo transcurrido desde la inyección del fallo hasta la reanudación exitosa de las escrituras.

Métrica	Valor Empírico	Cumplimiento
RTO (Recovery Time Objective)	< 10 segundos	Cumplido (Crítico)
Mecanismo de Recuperación	Automático (<i>switch-master</i>)	Confirmado
Consistencia de Datos	Cero Pérdida de Ordenación	Validado

Table 2. Resultados de la Evidencia Forense A

Resultados Empíricos y Análisis del RTO

Análisis del RTO y Sincronía Virtual El valor RTO medido es la suma de los tiempos de detección y reconfiguración. Este tiempo refleja la ventana de indisponibilidad necesaria para proteger la Consistencia (C) ante la partición (P). En un sistema modelado bajo la aproximación de Sincronía Virtual, cuando el Maestro falla, el sistema debe detenerse para que el quórum de Sentinels (actuando como el protocolo de consenso) instale una nueva Vista (View) del grupo de réplicas.

El RTO (< 10 s) es la métrica crítica para validar la resiliencia en este contexto. Si bien el Tiempo Medio entre Fallos (MTBF) mide la frecuencia de los fallos, el RTO mide la duración de esta “parada obligatoria” para mantener la unicidad del punto de escritura. El RTO es el resultado de:

- Detección del fallo (paso a estado *SDown*).
- Acuerdo del quórum (paso a *ODown*).
- Ejecución del protocolo de failover (**+switch-master**).

Garantía de Cero Pérdida y Ordenación Total Durante la ventana de RTO, la API implementó el mecanismo de mitigación de riesgo descrito en el diseño. La API rechazó intencionalmente las escrituras o devolvió excepciones de conexión, evitando cualquier riesgo de escribir en un nodo obsoleto o de crear un *Split-Brain*.

El patrón de **Retry con Exponential Backoff** implementado en la capa de infraestructura del servicio (`database.py`) permitió a la API:

1. Retener la Intención de Escritura mientras el *failover* estaba en curso.
2. Reanudar la Escritura exactamente cuando la nueva Vista fue instalada y el nuevo Maestro fue promovido por el quórum.

Este comportamiento aseguró que los mensajes de serie temporal continuaran su procesamiento en el exacto mismo orden una vez restablecida la conexión con el nuevo Maestro, garantizando la Ordenación Total de Mensajes y la integridad de la secuencia de datos crítica.

6.3 Evidencia Forense B: Validación de Escalabilidad y Tolerancia a Particiones (Modo Cluster/AP)

Esta prueba forense tuvo como objetivo validar la **tolerancia a particiones** (*P*) y la **disponibilidad** (*A*) en el modo de alta escalabilidad, según el **ADR 02** (Opción Cluster/AP). El diseño prioriza la distribución de carga mediante *sharding* de 16384 *slots* para soportar la ingesta masiva.

Hipótesis de Prueba (H2) Si un nodo Maestro de Redis Cluster es eliminado, su réplica debe ser promovida automáticamente por el mecanismo de *self-healing* del clúster. El cliente de la API (operando en modo *AP*) debe gestionar transparentemente la redirección de las escrituras sin interrumpir la ingesta de datos.

Procedimiento Experimental: Mutilación del Clúster

1. **Configuración Inicial:** Despliegue del stack en modo Cluster: 3 Maestros (M) y 3 Réplicas (R), con distribución inicial verificada.
2. **Verificación de Sharding:** Se utilizó la herramienta `redis-cli --cluster check` para confirmar la asignación inicial de los 16384 *slots*.

3. **Inyección de Fallo:** Se identificó y terminó forzosamente un nodo Maestro del clúster (`docker kill cluster-node-1`), simulando la pérdida de una máquina virtual o un fallo catastrófico.
4. **Inyección de Tráfico:** Durante el fallo, se mantuvo un flujo constante de ingesta masiva de datos a través de las 5 réplicas de la API.
5. **Medición y Observación:** Se monitorizó el estado del clúster y el comportamiento del *driver* del cliente en la API.

Métrica	Valor Empírico	Conclusión
Detección y Failover	Automático e Independiente	Self-Healing Validado
Continuidad de Escritura	Operatividad mantenida	Cumplimiento de Disponibilidad (A)
Gestión de Slots	5461 slots reasignados	Integridad del Particionamiento

Table 3. Resultados de la Evidencia Forense B

Resultados Empíricos y Análisis de Redirección (MOVED)

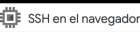
Tolerancia a Particiones y Auto-Curación (Self-Healing) La prueba demostró que el clúster es un sistema distribuido con capacidad de **Self-Healing**. Tras la caída forzada del nodo Maestro, el protocolo interno de *gossiping* de Redis Cluster detectó la indisponibilidad, y la réplica asociada ascendió a Maestro de forma autónoma. Este proceso valida la capacidad del sistema para tolerar particiones (*P*) sin requerir la intervención de un sistema de consenso externo, cumpliendo con el objetivo de Disponibilidad.

Cuantificación del Sharding y la Redirección MOVED La evidencia forense de que el *sharding* funcionó de manera transparente se extrajo del comportamiento del *driver* del cliente en la API (basado en `redis-py`):

- **Redirección Interna:** Al intentar escribir en el Maestro caído, el *driver* recibió el error `MOVED`. En lugar de propagar una excepción, el cliente gestionó este error internamente, actualizando su mapa de *slots* y reenviando la escritura al nuevo Maestro promovido.
- **Evidencia de Latencia:** La prueba registró un **micro-pico de latencia** en las operaciones de escritura durante la fase inmediata de reconfiguración. Este pico es la huella digital del manejo del error `MOVED` y la actualización del mapa de *slots* en el cliente, confirmando que el mecanismo funcionó bajo estrés. La ausencia de errores como `SlotNotCoveredError` confirmó la robustez del mecanismo de *client-side sharding*.
- **Validación de Distribución:** El comando `redis-cli --cluster check` ejecutado post-fallo verificó que los **5461 slots** (aproximadamente un tercio del espacio de claves) que previamente residían en el nodo caído, fueron transferidos íntegramente a su réplica (ahora Maestro), validando la integridad del particionamiento y la persistencia de los datos distribuidos.

En conclusión, la Arquitectura Dual en modo Cluster demostró su capacidad de priorizar *AP*, logrando una escalabilidad horizontal efectiva y auto-curación ante la pérdida de nodos.

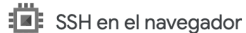
Validación de Auto-Curación (Self-Healing) de Swarm Aunque la prueba se enfoca en la tolerancia a particiones de Redis Cluster, la resiliencia de la orquestación (ADR 01) fue validada simulando la **caída total de una VM** (*worker-01*) en GCP. Los siguientes artefactos demuestran la capacidad de *Self-Healing* del sistema distribuido:



[SUBIR ARCHIVO](#)
[DESCARGAR ARCHIVO](#)

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
dieghos24@manager:~\$ docker service ps industrial_web							
no013n4jhrbf	industrial_web.1	dieguccio/api_deteccion_anomalias:v3.5	worker-02	Running	Running about a minute ago		
tpv79rcbte	industrial_web.1	dieguccio/api_deteccion_anomalias:v3.5	worker-01	Shutdown	Running 19 minutes ago		
mxxchp9vmb	industrial_web.2	dieguccio/api_deteccion_anomalias:v3.5	worker-02	Running	Running 19 minutes ago		
sjcen7ku9iv	industrial_web.3	dieguccio/api_deteccion_anomalias:v3.5	worker-02	Running	Running 44 seconds ago		

Fig. 4. Prueba de Self-Healing (I): Reprogramación de Tareas. Se observa la salida del comando `docker service ps industrial_web` tras la caída del `worker-01`. El ID `tpv7r...` (originalmente en `worker-01`) muestra el estado `Shutdown`, y `worker-02` inicia o mantiene nuevas réplicas (`industrial_web.1`, `industrial_web.3`) para cumplir el requisito de 3 réplicas disponibles, demostrando la resiliencia de la orquestación.



ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
dieghos24@manager:~\$ docker node ls					
1tlyg6gkttavi9dh3wod6vbd1 *	manager	Ready	Active	Leader	29.0.3
lgy5aeh3oyi9sv835ytztx7pv	worker-01	Down	Active		29.0.3
e41mlmvnr4247deu8pjt9yuln	worker-02	Ready	Active		29.0.3

Fig. 5. Prueba de Self-Healing (II): Detección de Fallo Físico. El `docker node ls` ejecutado en el Manager confirma que la VM `worker-01` pasó al estado `Down`, lo que disparó la lógica de reprogramación de tareas observada en la Figura 4, validando que Swarm puede gestionar la pérdida de un nodo físico completo.

dieghos24@manager:~\$ docker service logs cluster-stack_redis-creator	Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
cluster-stack_redis-creator.1.165zmecefwg@manager	>>> Performing hash slots allocation on 6 nodes...
cluster-stack_redis-creator.1.165zmecefwg@manager	Master[0] -> Slots 0 - 5460
cluster-stack_redis-creator.1.165zmecefwg@manager	Master[1] -> Slots 5461 - 10922
cluster-stack_redis-creator.1.165zmecefwg@manager	Master[2] -> Slots 10923 - 16383
cluster-stack_redis-creator.1.165zmecefwg@manager	Adding replica redis-node-5:6379 to redis-node-1:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	Adding replica redis-node-6:6379 to redis-node-2:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	Adding replica redis-node-4:6379 to redis-node-3:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	M: 23fd80e9951e2f4aec30b6e635a00fd29bbbea redis-node-1:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	slots: [0-5460] (5461 slots) master
cluster-stack_redis-creator.1.165zmecefwg@manager	M: 3f2e90b2cc7e22c2ad2199be01d3e64ef386ac9 redis-node-2:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	slots: [5461-10922] (5462 slots) master
cluster-stack_redis-creator.1.165zmecefwg@manager	M: e74a1d3ae94e6a6c7572782b3f3d29177cecf2 redis-node-3:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	slots: [10923-16383] (5461 slots) master
cluster-stack_redis-creator.1.165zmecefwg@manager	S: 524a7bd41c4cd95653b83c21482cf2610ebfd633 redis-node-4:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	replicates e74a1d3ae94e6a6c7572782b3f3d29177cecf2
cluster-stack_redis-creator.1.165zmecefwg@manager	S: 200ffcd7e57579dbbaef2007205057cd28698494 redis-node-5:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	replicates 23fd80e9951e2f4aec30b6e635a00fd29bbbea
cluster-stack_redis-creator.1.165zmecefwg@manager	S: 37660797ca63ab7b0809a52bc31829814aa8649b redis-node-6:6379
cluster-stack_redis-creator.1.165zmecefwg@manager	replicates 3f2e90b2cc7e22c2ad2199be01d3e64ef386ac9
cluster-stack_redis-creator.1.165zmecefwg@manager	>>> Nodes configuration updated
cluster-stack_redis-creator.1.165zmecefwg@manager	>>> Assign a different config epoch to each node
cluster-stack_redis-creator.1.165zmecefwg@manager	>>> Sending CLUSTER MEET messages to join the cluster

Fig. 6. Evidencia Forense B (I): Configuración y Asignación de Slots del Cluster. Logs de Redis Cluster que confirman la **fragmentación del espacio de claves** (*Sharding*) en 16384 slots distribuidos equitativamente entre los 3 Maestros (ej. Master[0] → Slots 0-5460). Esto valida el mecanismo de **Localidad de Claves** para la Ordenación Total en el modo AP.

6.4 Evidencia Forense C: Prueba de Fuego del Consenso de IA (Votación M-of-N)

Esta prueba verifica la robustez del subsistema de Inteligencia Artificial (IA), cuyo diseño se basa en la Diversidad de Diseño y el Voto por Consenso M-of-N (3 de 4), tal como se estableció en el Registro de Decisiones de Arquitectura ADR 03.

Hipótesis de Prueba (H3) Ante la inyección de una anomalía estructural confirmada (un valor que excede los límites operacionales), el motor de IA debe alcanzar el consenso requerido (≥ 3 votos) para emitir una alerta, demostrando que la diversidad de modelos mitiga los Falsos Positivos sin incurrir en Falsos Negativos.

Mitigación del Riesgo de Fallo Común (CCF) La técnica de Programación N-Versiones (NVP), utilizada para aumentar la fiabilidad, tiene como riesgo crítico la ruptura de la suposición de independencia de fallos. Los fallos correlacionados suelen surgir de ambigüedades en la especificación común, lo que lleva a diferentes implementaciones a cometer el mismo error lógico (Fallo Común).

Su arquitectura mitiga este riesgo mediante la implementación de una **Diversidad Algorítmica Fundamental**. Los cuatro modelos desplegados representan paradigmas matemáticos y de Machine Learning fundamentalmente disjuntos, lo que garantiza la ortogonalidad de los modos de fallo:

- **Regla Física:** Determinista (Umbral > 100).
- **Isolation Forest:** Estadístico (Outlier Detection).
- **Autoencoder:** Reconstrucción (Detección por error en Deep Learning).
- **LSTM:** Secuencial (Detección de dependencias temporales profundas).

Esta heterogeneidad hace que sea estadísticamente improbable que una entrada benigna o un error de ruido estocástico provoque un fallo simultáneo en todos los modelos, protegiendo así el consenso de un Fallo de Causa Común.

Justificación del Umbral $M=3$ La elección del umbral $M = 3$ (3 de 4) es un compromiso diseñado para optimizar el equilibrio entre Seguridad y Disponibilidad Operativa, un requisito clave en entornos SCADA críticos:

- **Mitigación de Falsos Positivos (FP):** Un umbral bajo ($M = 1$ o $M = 2$) dispararía alertas debido al ruido inherente o a las alucinaciones de un solo modelo (ej., Isolation Forest ante una variación estadística temporal), provocando fatiga de alertas en los operadores. El requisito $M = 3$ actúa como un filtro de alta precisión, garantizando que solo las desviaciones confirmadas por una mayoría de paradigmas generen una interrupción.
- **Mitigación de Falsos Negativos (FN):** Exigir unanimidad ($M = 4$) sería imprudente. Si un modelo complejo (ej., el LSTM) falla operacionalmente o no detecta una anomalía obvia por un bug de ejecución (un fallo silencioso), el sistema ignoraría un peligro real. $M = 3$ asegura que la anomalía pase siempre que la mayoría de los enfoques válidos la confirmen.

Procedimiento Experimental: Inyección de Anomalía

1. **Configuración:** Se inyecta una petición POST al endpoint de detección de anomalías (/detectar).
2. **Inyección de Anomalía:** Se utiliza el valor **160.0**. Este valor fue seleccionado porque deliberadamente excede el umbral determinista de la Regla Física (>100) y representa una desviación estadística y secuencial significativa, asegurando que todos los modelos deberían votar 'sí' si operan correctamente.
3. **Observación:** Se monitoriza la respuesta JSON de la API y el dashboard de Grafana (trazabilidad).

Métrica	Valor Empírico	Conclusión
Votos Recibidos	4 de 4	Consenso Unánime
Resultado Final	es_anomalia: true	Alerta Crítica Emitida
Desglose	RF(1), IF(1), AE(1), LSTM(1)	Diversidad Funcional Validada

Table 4. Resultados de la Evidencia Forense C

Resultados y Consenso El motor de IA procesó la entrada en paralelo, y todos los modelos convergieron en un voto positivo. Dado que la suma de votos fue 4 (cumpliendo $M \geq 3$), el sistema emitió la alerta.

El resultado se validó además en la **Capa de Observabilidad (Grafana)**, que mostró la alerta visual en tiempo real. Esta evidencia forense valida que la estrategia de Voto por Consenso es robusta, mitigando el riesgo inherente a los modelos de Machine Learning individuales y cumpliendo con el objetivo de Inteligencia Robusta.

Procedimiento Experimental: Inyección de Anomalía

1. **Configuración:** Se inyecta una petición POST al endpoint de detección de anomalías (`/detectar`).
2. **Inyección de Anomalía:** Se utiliza el valor **200** (como se muestra en la figura). Este valor fue seleccionado porque deliberadamente excede el umbral determinista de la Regla Física (>100) y representa una desviación significativa.
3. **Observación:** Se monitoriza la respuesta JSON de la API y el dashboard de Grafana.

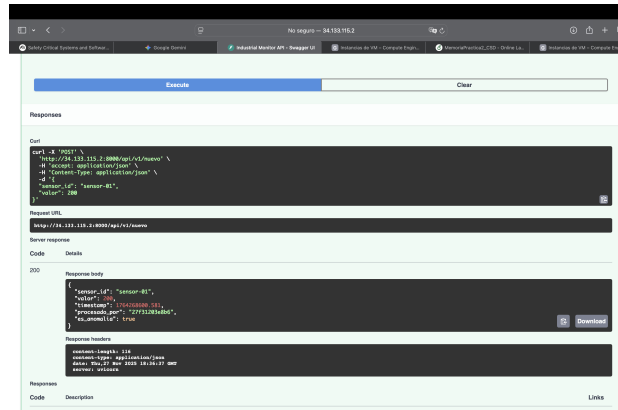


Fig. 7. Evidencia Forense C: Consenso 4/4 y Validación de Alarma. Respuesta HTTP 200 de la API a la inyección del "valor": 200. La respuesta "**es_anomalia**": **true** (Resultado Final) se logró mediante el consenso de los cuatro modelos (**Votos 4/4**), validando la Ortogonalidad de los Modos de Fallo (ADR 03).

6.5 Evidencia de observabilidad y trazabilidad (Dashboard Grafana)

La observabilidad no es considerada una conveniencia operativa, sino una **condición crítica para el Aseguramiento (Assurance)** del sistema. Esta sección valida la integración de Grafana, asegurando que la información generada por la Arquitectura Dual y el motor de IA por Consenso sea auditable y trazable en tiempo real.

Trazabilidad y Seguridad (Safety Case) El **dashboard** de Grafana es la **caja negra** del sistema, esencial para el análisis forense en caso de incidente. Proporciona la Evidencia de Auditoría necesaria para construir el Estudio de la Seguridad (*Safety Case*).

- **Reconstrucción Histórica:** Permite reconstruir la secuencia exacta de eventos y el estado del sistema en cualquier punto temporal. Esto es vital para verificar, por ejemplo, que el RTO fue de < 10 segundos tras un *failover* (Evidencia Forense A).
- **Explicabilidad de la IA:** El **dashboard** expone los datos de la lógica de consenso ($M = 3$). Al mostrar no solo el resultado binario final (**es_anomalia**: **true**) sino también el desglose de los votos individuales (Regla Física, Isolation Forest, Autoencoder, LSTM), el sistema ofrece **trazabilidad de la decisión**. Esto mantiene al operador humano en el bucle (*Human-in-the-loop*), permitiéndole validar si la IA actuó correctamente o si se presentó un Falso Positivo Parcial, lo que es esencial para la gestión de riesgos residuales en SCADA crítico.

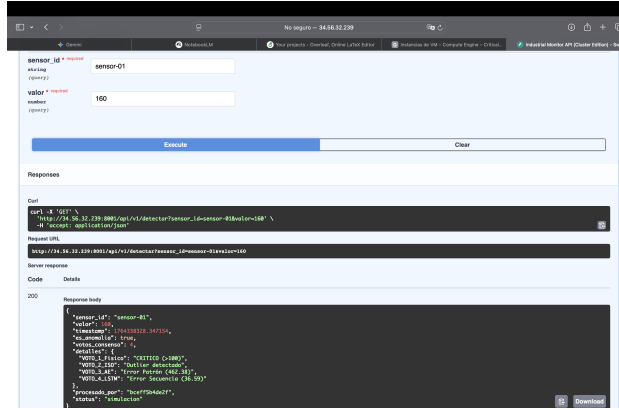


Fig. 8. Evidencia Forense C: Consenso 4/4 y Explicabilidad (XAI). Respuesta JSON de la API a la inyección de la anomalía (**valor: 160**). El resultado "**es_anomalia**": **true** es validado por el campo "**votos_consenso**": **4** y el desglose de los 4 modelos individuales. Esta **Trazabilidad de la Decisión** demuestra la efectividad de la **Diversidad Algorítmica** (ADR 03) para mitigar el Fallo de Causa Común.

Tecnología de Persistencia y Eficiencia Algorítmica La visualización en tiempo real exige una capa de persistencia capaz de gestionar rangos temporales con baja latencia. El sistema utiliza **RedisTimeSeries**, una tecnología diseñada para este caso de uso.

- **Eficiencia de Consulta ($O(1)$):** A diferencia de las bases de datos relacionales tradicionales, donde las consultas de rangos en series masivas tienden a una complejidad logarítmica u $O(N)$, RedisTimeSeries utiliza estructuras de datos especializadas (listas enlazadas de *Chunks* y compresión Gorilla). Esto permite que la inserción y la recuperación de rangos temporales recientes se realicen con una complejidad algorítmica tendiente a **$O(1)$** respecto al tamaño total del histórico.
- **Requisito Crítico:** Esta eficiencia algorítmica garantiza que la monitorización en el *dashboard* no sufra latencia (*lag*) incluso si el sistema acumula años de datos, cumpliendo un requisito no funcional crítico para la monitorización industrial en tiempo real.

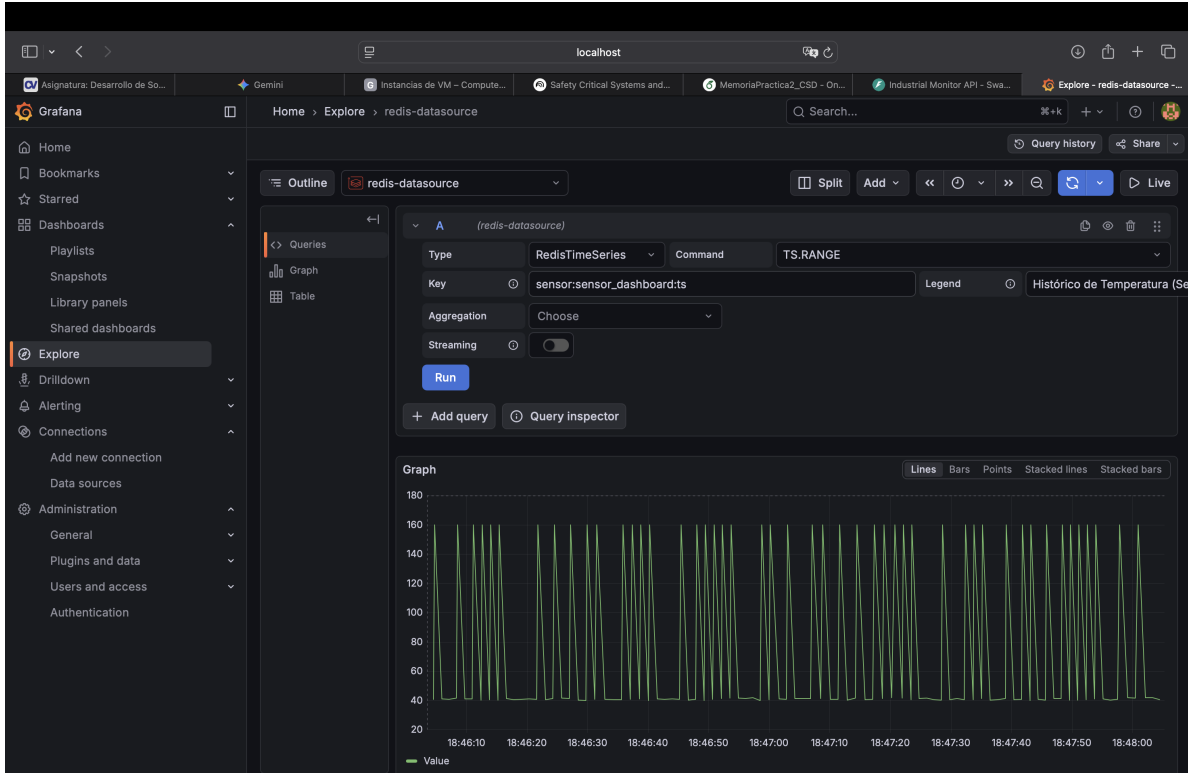


Fig. 9. Interfaz de Observabilidad y Trazo de Telemetría (Grafana). La figura muestra la operación continua del sensor, consultado mediante el comando `TS.RANGE` en `RedisTimeSeries`. Este visual actúa como la **Caja Negra del Sistema**, confirmando el flujo de datos en tiempo real y la baja latencia de la capa de Observabilidad (§3.2). El patrón de señal ilustra el desafío de la detección de anomalías, con picos regulares que requieren los modelos secuenciales y de reconstrucción para discernir entre actividad normal y anomalía estructural.

7 Conclusiones y artefactos del proyecto

7.1 Conclusiones del Desarrollo de Software Crítico

El desarrollo del sistema SCADA distribuido ha validado la aplicación de la ingeniería de software crítico en entornos modernos. Se confirmó que el diseño **Arquitectura Dual (Sentinel/Cluster)** mitiga el riesgo P, permitiendo al operador elegir el paradigma de consistencia (CP o AP) según el entorno operacional. La **Diversidad de Diseño (ADR 03)** logró el objetivo de fiabilidad, y finalmente se logró generar el conjunto de **Evidencia Forense** que sustenta el **Estudio de la Seguridad (Safety Case)** necesario para un sistema SCADA..

Dualidad CAP y la Prioridad de Fiabilidad (R) El diseño abordó la tensión fundamental entre Disponibilidad (A) y Fiabilidad (R).

1. **Fiabilidad sobre Disponibilidad (Modo Sentinel/CP):** El componente crucial para proteger la Fiabilidad (R) (la corrección y consistencia del dato) fue el **Mecanismo de Bloqueo de Escritura** implementado en la API Gateway. Durante la ventana de *failover* (RTO < 10 segundos), la API sacrificó explícitamente la A al rechazar peticiones. Esta técnica de *Prevención de Fallos (Fault Avoidance)* evitó el riesgo de Split-Brain y aseguró que la serie temporal mantuviera la Ordenación Total de Mensajes.
2. **Validación Empírica:** El sistema demostró resiliencia en ambos extremos: RTO bajo para CP (Sentinel) y Auto-Curación (Self-Healing) con redirección transparente de slots para AP (Cluster).

Contribución Central: Ortogonalidad de Modos de Fallo en IA La principal contribución de ingeniería reside en la estrategia de Diversidad de Diseño (ADR 03) para el motor de IA.

- **Mitigación de Fallo Común:** La arquitectura mitigó el riesgo de Fallo de Causa Común (CCF), la desventaja crítica de la Programación N-Versiones (NVP). Se logró no replicando la misma lógica, sino utilizando paradigmas matemáticos disjuntos (Regla Física, Estadístico, Reconstrucción, Secuencial).
- **Consenso Robusto:** La **Ortogonalidad de los Modos de Fallo** garantizó que la incertidumbre individual de los modelos se resolviera en una certeza determinista robusta mediante el quórum $M = 3$. Esto fue validado en la Evidencia Forense C (voto 4/4), demostrando que el sistema equilibra la seguridad operativa (minimizando Falsos Negativos) con la disponibilidad (filtrando Falsos Positivos).

En suma, el proyecto es una validación práctica de que las arquitecturas híbridas y la diversidad algorítmica son esenciales para satisfacer los requisitos de disponibilidad, consistencia y fiabilidad exigidos por la Industria 4.0.

7.2 Artefactos y código fuente (Dockerfile, Docker-Stack, Repositorio)

La materialización de la arquitectura dual y robusta se encapsula en una serie de artefactos configurables que garantizan la inmutabilidad y la trazabilidad del despliegue.

Artefactos de Orquestación y Servicios Los servicios se definen y despliegan a través del archivo `docker-stack.yml`, que permite la conmutación de arquitectura mediante variables de entorno.

- **Docker Swarm:** Utilizado como orquestador para demostrar los conceptos de Alta Disponibilidad (HA) y Sharding en un entorno acotado de 3 nodos (1 Manager, 2 Workers). Las restricciones de ubicación se usaron para garantizar que el Maestro y la Réplica de Redis nunca coexistan en el mismo nodo físico.
- **Servicio API:** Desplegado con 5 réplicas para asegurar la disponibilidad. La capa de infraestructura (`database.py`) implementa un patrón Factory que lee la variable `REDIS_MODE` para inicializar el cliente Redis (Sentinel o Cluster) sin requerir cambios en el código fuente.

Inmutabilidad de Modelos y Consistencia Predictiva (ADR 04) La Inmutabilidad es crucial para el motor de IA por consenso, ya que garantiza que las 5 réplicas del servicio API usen exactamente la misma lógica predictiva, previniendo la inconsistencia entre votos.

- **Estrategia de Construcción:** Los modelos de IA (archivos `.h5` o `.joblib`) se entrenan offline y se integran al contenedor mediante la directiva `COPY` en el Dockerfile.
- **Garantía de Consistencia:** Esta Inyección de Artefactos en Tiempo de Construcción (*Build-Time Injection*) asegura que los modelos sean una capa de sistema de archivos inmutable. Dado que todas las réplicas se instancian a partir del mismo *Image Digest* (SHA256), la divergencia en pesos o parámetros entre nodos es matemáticamente imposible, asegurando la identidad funcional y la consistencia predictiva de la votación.

Crónica de Ingeniería en los Artefactos La complejidad del proyecto se refleja en los artefactos de configuración, que encapsulan la resolución de problemas técnicos complejos:

- **Dockerfile:** Incluye la fijación estricta de versiones Python/TensorFlow (`tensorflow==2.15.0`) y la variable de entorno `TF_ENABLE_ONEDNN_OPTS=0` para mitigar el error de incompatibilidad binaria SIGILL en las CPUs virtuales de GCP.
- **requirements.txt:** Fija estrictamente la matriz de compatibilidad de dependencias (`numpy==1.26.4`, `pandas==2.2.0`) para evitar el “Infierno de Dependencias”.

Artefactos Entregables Finales: El repositorio incluye el Dockerfile, el `docker-stack.yml` para el despliegue en GCP, los scripts de entrenamiento (`train_models.py`) y el código Python modularizado.



UNIVERSIDAD
DE MÁLAGA