

# Especificación de API Python - Universal Predictor

Ingeniería de Software

February 18, 2026

## Contents

<b>1</b>	<b>Introducción</b>	<b>3</b>
<b>2</b>	<b>Estructuras de Datos (Tipado)</b>	<b>3</b>
2.1	Configuración ( $\Lambda$ )	3
2.2	Entrada Operativa ( $y_t, y_{target}, \tau$ )	3
2.3	Salida del Sistema	3
<b>3</b>	<b>Arquitectura Multitenencia (Stateless/Functional Pattern)</b>	<b>4</b>
3.1	Maximización de Throughput (Batching Vectorizado)	4
<b>4</b>	<b>Clase Principal: UniversalPredictor (Stateful Wrapper)</b>	<b>5</b>
4.1	Inicialización	5
4.2	Método de Ejecución (Paso $t \rightarrow t + 1$ )	5
<b>5</b>	<b>Prevención de Fragmentación de VRAM (JAX Memory Management)</b>	<b>6</b>
<b>6</b>	<b>Persistencia (Atomic Snapshotting)</b>	<b>9</b>
<b>7</b>	<b>I/O Asíncrono para Snapshots (Non-Blocking)</b>	<b>10</b>
<b>8</b>	<b>Apagado Elegante (Graceful Shutdown) para Contenedores</b>	<b>13</b>
<b>9</b>	<b>Ajuste Adaptativo del Umbral CUSUM</b>	<b>20</b>
9.1	Fórmula de Ajuste	20
9.2	Interpretación de Curtosis	20
<b>10</b>	<b>Periodo de Gracia (Ventana Refractaria) Post-Cambio de Régimen</b>	<b>20</b>
10.1	Motivación	20
10.2	Solución: Grace Period (Refractario)	21
10.3	Algoritmo de Implementación	21
10.4	Parámetros Sugeridos	22
10.5	Diagnóstico y Telemetría	22
<b>11</b>	<b>Flags de Operación y Recuperación</b>	<b>22</b>
11.1	DegradedInferenceMode	22
11.2	EmergencyMode	22
11.3	RegimeChangeDetected	22
11.4	ModeCollapseWarning	22
11.5	Período de Gracia (Refractario) Post-Cambio de Régimen	23
11.5.1	Implementación	23
11.5.2	Dinámica Temporal	23
11.5.3	Parámetros Recomendados	24
11.5.4	Ventajas	24

<b>12 Manejo de Errores y Excepciones</b>	<b>24</b>
12.1 Excepciones Estándar . . . . .	24
12.2 Alertas Específicas Avanzadas . . . . .	24
12.3 Ejemplo de Logging en Producción . . . . .	25
<b>13 Detección de Mode Collapse en DGM</b>	<b>26</b>
13.1 Criterio de Detección . . . . .	26
13.2 Acción Correctiva . . . . .	26
<b>14 Determinismo de Punto Flotante (Bit-Exact Reproducibility)</b>	<b>26</b>
<b>15 Load Shedding Dinámico (Poda Topológica en Cisne Negro)</b>	<b>28</b>
<b>16 Telemetría de Jitter (Gatillo Preciso para Load Shedding)</b>	<b>32</b>

# 1 Introducción

Este documento detalla la implementación en Python de la interfaz abstracta I/O definida en *Predictor\_Estocastico\_IO*. La API expone la clase `UniversalPredictor`, diseñada para entornos de alto rendimiento utilizando JAX para la aceleración numérica.

## 2 Estructuras de Datos (Tipado)

Se utilizan `dataclasses` y `jaxtyping` para garantizar la inmutabilidad y el tipado dimensional estricto de los tensores.

### 2.1 Configuración ( $\Lambda$ )

```
1 from dataclasses import dataclass
2 from typing import Optional
3 from jaxtyping import Float, Array, Bool
4
5 @dataclass(frozen=True)
6 class PredictorConfig:
7     """Vector de Hiperparámetros Lambda."""
8     schema_version: str = "1.0" # Versionado de snapshots (evita incompatibilidades)
9     epsilon: float = 1e-3 # Regularización Entrópica (Sinkhorn)
10    learning_rate: float = 0.01 # Tasa de Aprendizaje JKO
11    log_sig_depth: int = 3 # Profundidad de Firma (Kernel D)
12    wtmm_buffer_size: int = 128 # Memoria WTMM (N_buf)
13    besov_cone_c: float = 1.5 # Cono de Influencia de Besov
14    holder_threshold: float = 0.4 # Umbral Circuit Breaker (H_min)
15    cusum_h: float = 5.0 # Umbral Drift (h)
16    cusum_k: float = 0.5 # Slack (k)
17    grace_period_steps: int = 20 # Período refractario post-cambio régimen (silencia CUSUM)
18    volatility_alpha: float = 0.1 # Decaimiento EWMA de Varianza
19
20    # Política de Abandono y Anti-Aliasing
21    staleness_ttl_ns: int = 500_000_000 # TTL Latencia (500ms)
22    besov_nyquist_interval_ns: int = 100_000_000 # Límite Nyquist (100ms) para estabilidad WTMM
23    inference_recovery_hysteresis: float = 0.8 # Factor histéresis para recuperación de modo degradado
```

### 2.2 Entrada Operativa ( $y_t, y_{target}, \tau$ )

```
1 @dataclass(frozen=True)
2 class MarketObservation:
3     price: Float[Array, "1"] # y_t (Normalizado o Absoluto)
4     target: Float[Array, "1"] # y_target (Generalmente price actual)
5     timestamp_ns: int # Unix Epoch (Nanosegundos)
6
7     def validate_domain(self, sigma_bound: float = 20.0, sigma_val: float = 1.0) -> bool:
8         """Detección de Outliers Catastróficos (> 20 sigma)."""
9         return abs(self.price) <= (sigma_bound * sigma_val)
```

### 2.3 Salida del Sistema

```
1 @dataclass(frozen=True)
2 class PredictionResult:
3     predicted_next: Float[Array, "1"] # y_{t+1} (Espacio Z-Score)
4
5     # Telemetría de Estado (S_risk)
6     holder_exponent: Float[Array, "1"] # H_t
7     cusum_drift: Float[Array, "1"] # G^+
8     distance_to_collapse: Float[Array, "1"] # h - G^+
9     free_energy: Float[Array, "1"] # F (Energía JKO)
10
11    # Telemetría Avanzada (Nuevas Adiciones)
12    kurtosis: Float[Array, "1"] # _t - Curtosis empírica de errores
```

```

13 dgm_entropy: Float[Array, "1"] # H_DGM - Entropía del predictor DGM (NaN si inactivo)
14 adaptive_threshold: Float[Array, "1"] # h_t - Umbral CUSUM adaptativo
15
16 # Estado del Orquestador
17 weights: Float[Array, "4"] # [rho_A, rho_B, rho_C, rho_D] (Simplex)
18
19 # Flags de Salud y Control (Explícitos)
20 sinkhorn_converged: Bool[Array, "1"] # Convergencia JKO
21 degraded_inference_mode: bool # TTL violation (congelamiento de pesos)
22 emergency_mode: bool # H_t < H_min (singularidad crítica)
23 regime_change_detected: bool # CUSUM alarm (G+ > h_t)
24 mode_collapse_warning: bool # H_DGM < ·H[g] (colapso DGM)
25
26 mode: str # "Standard" | "Robust" | "Emergency"

```

### 3 Arquitectura Multitenencia (Stateless/Functional Pattern)

Para soportar cientos de activos (Multi-Asset) en un solo servidor, la API soporta un modo puramente funcional. Esto permite gestionar el estado en bases de datos externas de baja latencia (Redis) y compartir el grafo de computación JAX compilado (el Predictor) entre todos los activos.

#### 3.1 Maximización de Throughput (Batching Vectorizado)

Esta arquitectura habilita el uso de `jax.vmap` para procesar lotes de estados de múltiples activos en una sola llamada al hardware, minimizando el impacto del GIL de Python y maximizando la ocupación de la GPU.

```

1 class FunctionalPredictor:
2     """
3     Implementación Stateless para JAX Core.
4     Permite escalar a miles de predictores compartiendo la misma estructura computacional.
5     """
6
7     def __init__(self, config: PredictorConfig):
8         # Compilación JIT única para todos los activos
9         # Habilita vectorización automática (vmap) sobre la dimensión del batch (activos)
10        self.config = config
11        self._core_step = self._core_update_step
12        self._jit_update = jax.jit(self._core_step)
13        self._vmap_update = jax.jit(jax.vmap(self._core_step, in_axes=(0, 0, 0, 0)))
14
15    def init_state(self):
16        """Genera un estado cero inicial (cold state structure)."""
17        return self._initialize_state_structure()
18
19    def step(self, state, obs: MarketObservation) -> tuple[object, PredictionResult]:
20        """
21        Transición de Estado Pura: (S_t, Obs_t) -> (S_{t+1}, Pred_{t+1})
22        """
23        # 1. Validaciones (Outlier, Staleness, Nyquist) logic idéntica a UniversalPredictor
24        # ... logic for freeze_weights flag calculation ...
25
26        # 2. Ejecución Kernel JAX
27        # Zero-Copy: La actualización de búferes ocurre dentro de XLA (dynamic_update_slice)
28        new_state, raw_result = self._jit_update(
29            state, # Estado inyectado explícitamente desde Redis/Memoria
30            obs.price,
31            obs.target,
32            freeze_weights=should_freeze
33        )
34
35        # 3. Mapeo de Resultados
36        result = PredictionResult(
37            predicted_next=raw_result.y_next,
38            # ... resto de campos ...
39        )
40
41        return new_state, result
42

```

```

43 def step_batch(self, states, obs_batch: MarketObservation):
44     """
45     Procesamiento vectorizado para N activos simultáneos.
46     Utiliza vmap para paralelizar la inferencia y actualización.
47     """
48     # ... logic for batch flags ...
49     new_states, results = self._vmap_update(states, obs_batch.price, obs_batch.target,
50     freeze_flags)
51     return new_states, results

```

## 4 Clase Principal: UniversalPredictor (Stateful Wrapper)

Esta clase envuelve el patrón funcional para casos de uso de un solo activo (Single-Tenant), manteniendo el estado en memoria local (`self._state`).

### 4.1 Inicialización

```

1 class UniversalPredictor:
2     def __init__(self, config: PredictorConfig):
3         """
4         Inicializa el grafo de cómputo JAX (XLA JIT compilation).
5         Asigna memoria estática para los búferes en el dispositivo (VRAM).
6         El estado interno (self._state) contiene los `jnp.array` persistentes (rolling buffers
7         )
8         que se actualizarán mediante operaciones funcionales (jnp.roll, lax.dynamic_update)
9         para eliminar la latencia de transferencia de memoria (Zero-Copy).
10        """
11        self.config = config
12        self._state = self._initialize_state() # Estado interno JAX (resident un GPU)
13        self._jit_update = jax.jit(self._core_update_step)
14        self._last_timestamp_ns = 0 # Para cálculo de frecuencia
15
16    def fit_history(self, history: list[float]) -> bool:
17        """
18        Bootstrapping inicial (Protocolo de Cold Start).
19        Procesa el lote histórico para estabilizar los pesos JKO y llenar los búferes.
20        Requiere un mínimo de N_buf muestras.
21
22        Returns:
23            bool: True si el sistema alcanzó convergencia estable (Sinkhorn + CUSUM).
24        Raises:
25            ValueError: Si el historial es insuficiente (< wtm_buffer_size).
26            RuntimeError: Si el sistema diverge tras el calentamiento.
27        """
28        if len(history) < self.config.wtm_buffer_size:
29            raise ValueError(f"Historial insuficiente. Requerido: {self.config.
30            wtm_buffer_size}")
31
32        # Ejecución batch acelerada (jax.lax.scan) para calentar el estado
33        # Simula el paso del tiempo para llenar colas y estabilizar gradientes
34        self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))
35
36        # Validación de Convergencia
37        is_converged = final_metrics.sinkhorn_converged
38        is_stable = final_metrics.cusum_drift < self.config.cusum_h
39
40        if not (is_converged and is_stable):
41            logger.warning("Cold Start finalizado sin convergencia estable.")
42            return False
43
44        return True

```

### 4.2 Método de Ejecución (Paso $t \rightarrow t + 1$ )

```

1 def step(self, obs: MarketObservation) -> PredictionResult:
2     """
3     Ejecuta un ciclo completo de predicción.
4     Maneja internamente la validación de dominio y TTL.

```

```

5      """
6      # 1. Validación de Dominio (Outlier Check)
7      if not obs.validate_domain():
8          logger.error("Outlier Catastrófico detectado. Ignorando tick.")
9          return self._last_valid_result # Mantiene inercia
10
11     # 2. Check de Abandono (Staleness) y Frecuencia (Anti-Aliasing)
12     current_time = time.time_ns()
13     latency = current_time - obs.timestamp_ns
14     is_stale = latency > self.config.staleness_ttl_ns
15
16     # Validación de Frecuencia Nyquist (WTMM Stability)
17     dt_arrival = obs.timestamp_ns - self._last_timestamp_ns
18     is_sparse = (self._last_timestamp_ns > 0) and (dt_arrival > self.config.
besov_nyquist_interval_ns)
19
20     if is_sparse:
21         logger.warning(f"FrequencyWarning: Event interval {dt_arrival}ns > Nyquist limit.
WTMM spectrum might alias.")
22
23     self._last_timestamp_ns = obs.timestamp_ns
24
25     # 3. Actualización Core (JAX) - Zero-Copy State Management
26     # IMPORTANTE: El buffer de señal reside en GPU/TPU (self._state.signal_buffer).
27     # La actualización se realiza "in-place" funcionalmente usando jax.lax.
dynamic_update_slice
28     # o jnp.roll dentro del kernel compilado para evitar transferencias CPU <-> VRAM.
29     # Si hay staleness o sparsity excesiva, se congelan pesos para no degradar la geometrí
a.
30     should_freeze = is_stale or is_sparse
31
32     new_state, result_data = self._jit_update(
33         self._state,
34         obs.price,
35         obs.target,
36         freeze_weights=should_freeze,
37         # No se pasa history_buffer explícitamente, ya vive en _state
38     )
39
40     self._state = new_state
41
42     # 4. Empaquetado de Resultados
43     return PredictionResult(
44         predicted_next=result_data.y_next,
45         holder_exponent=result_data.H_t,
46         sinkhorn_converged=result_data.converged,
47         is_stable=not (is_stale or is_sparse),
48         # ... mapeo resto de campos
49     )

```

## 5 Prevención de Fragmentación de VRAM (JAX Memory Management)

**Problema de Producción:** JAX preasigna el 90% de la memoria GPU (VRAM) mediante el runtime XLA en el primer acceso, bajo el modelo *single-GPU-device-per-process*. En sistemas de alta disponibilidad con ejecución continua, la fragmentación de memoria puede acumularse tras semanas de operación, causando **Out Of Memory (OOM)** silencioso o degradación de rendimiento.

**Escenario:**

1. Proceso inicia y JAX asigna ~90% VRAM (ej. 36/40 GB en una GPU A100).
2. Durante  $N$  horas, se crean/liberan tensores temporales en el algoritmo de Sinkhorn, WTMM, DGM.
3. El recolector de basura de Python libera memoria CPython, pero XLA mantiene fragmentos aislados.
4. Tras ~1-4 semanas: OOM silencioso en operación crítica (pérdida de predicción).

## Solución: Control Granular de Asignación de VRAM

Configurar dos variables de entorno críticas **ANTES** de importar JAX:

```
1 import os
2
3 # PASO 1: Limitar asignación inicial de VRAM
4 # Por defecto JAX asigna 90% → reservar solo 70% para dejar margen
5 os.environ['XLA_PYTHON_CLIENT_MEM_FRACTION'] = '0.7'
6
7 # PASO 2: Usar allocador "platform" para liberar dinámicamente si Python GC lo exige
8 # Opciones:
9 #   'platform' (recomendado): Libera memoria al solicitar el SO
10 #   'bfc' (default): Caché de bloques fija (menos flexible)
11 os.environ['XLA_PYTHON_CLIENT_ALLOCATOR'] = 'platform'
12
13 # PASO 3: Habilitar protección contra fragmentación
14 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true' # Permite crecimiento incremental
15
16 # Ahora importar JAX (después de fijar variables de entorno)
17 import jax
18 import jax.numpy as jnp
19
20 print(f"VRAM allocation: {jax.devices()}")
21 print(f"XLA allocator: platform (dynamic)")
```

### Implicaciones de Configuración:

Variable	Default	Recomendado	Efecto
XLA_PYTHON_CLIENT_MEM_FRACTION	0.9	0.7	Deja 30% libre para I/O, SO, buffer
XLA_PYTHON_CLIENT_ALLOCATOR	bfc	platform	Libera dinámicamente al GC
TF_FORCE_GPU_ALLOW_GROWTH	false	true	Crecimiento incremental (no prealloc)

### Análisis de VRAM Disponible:

Asumiendo GPU A100 (40 GB):

- **Sin configuración:** JAX asigna 36 GB → SO + threads + buffers tienen 4 GB
- **Con configuración:** JAX asigna 28 GB → margen de 12 GB para overhead
- Sistema más estable: mejor para operaciones fragmentadas (Sinkhorn iterativo, WTMM multiscala)

### Mitigación de Fragmentación: Estrategia de Pooling

Además de las variables de entorno, implementar *pool* de tensores pre-asignados para operaciones críticas:

```
1 class VRAM_PooledAllocator:
2     def __init__(self, device_memory_budget_gb: float = 28.0, pool_size: int = 100):
3         """
4         Crea un pool de tensores pre-asignados para reducir fragmentación.
5         """
6         self.device_budget_bytes = device_memory_budget_gb * 1e9
7         self.pool = []
8         self.available = []
9
10        # Pre-asignar tensores comunes (ej. buffers de Sinkhorn)
11        for i in range(pool_size):
12            tensor = jnp.zeros((1024, 1024), dtype=jnp.float32) # ~4MB
13            self.pool.append(tensor)
14            self.available.append(True)
15
16        def acquire_tensor(self, shape, dtype=jnp.float32):
17            """Obtiene tensor del pool sin fragmentar VRAM."""
18            for idx, available in enumerate(self.available):
19                if available:
20                    self.available[idx] = False
21                    return self.pool[idx]
22
23            # Si no hay disponible, crear temporalmente
24            return jax.device_put(jnp.zeros(shape, dtype=dtype))
25
26        def release_tensor(self, idx):
```

```

27         """Devuelve tensor al pool."""
28         if idx < len(self.available):
29             self.available[idx] = True
30
31     def memory_utilization_percent(self):
32         """Reporta fragmentación."""
33         used = sum(1 for av in self.available if not av)
34         return 100.0 * used / len(self.available)

```

### Monitoreo de Fragmentación:

```

1 import psutil
2 import subprocess
3
4 def monitor_vram_fragmentation(interval_seconds=60):
5     """
6     Thread de monitoreo que reporta fragmentación de VRAM.
7     """
8     import time
9     import threading
10
11     def _monitor():
12         while True:
13             try:
14                 # Consultar nvidia-smi para obtener uso real
15                 result = subprocess.run(
16                     ['nvidia-smi', '--query-gpu=memory.used,memory.total',
17                     '--format=csv,noheader'],
18                     capture_output=True, text=True, timeout=5
19                 )
20
21                 if result.returncode == 0:
22                     used, total = map(float, result.stdout.strip().split(','))
23                     utilization = 100.0 * used / total
24
25                     if utilization > 0.95:
26                         print(f"[WARNING] VRAM near saturation: {utilization:.1f}%")
27                     elif utilization > 0.85:
28                         print(f"[INFO] VRAM utilization: {utilization:.1f}% (elevated)")
29
30                     time.sleep(interval_seconds)
31             except Exception as e:
32                 print(f"[ERROR] VRAM monitoring failed: {e}")
33                 break
34
35     thread = threading.Thread(target=_monitor, daemon=True)
36     thread.start()

```

### Configuración de Despliegue Recomendada:

```

1 #!/bin/bash
2 # deployment/run_predictor.sh
3
4 # Variables de entorno CRÍTICAS para producción
5 export XLA_PYTHON_CLIENT_MEM_FRACTION=0.7
6 export XLA_PYTHON_CLIENT_ALLOCATOR=platform
7 export TF_FORCE_GPU_ALLOW_GROWTH=true
8
9 # Logs de configuración
10 echo "[INFO] XLA VRAM Fraction: 0.7 (28/40 GB en A100)"
11 echo "[INFO] Allocator: platform (dynamic)"
12 echo "[INFO] GPU growth: enabled"
13
14 # Ejecutar predictor
15 python3 -u predictor_service.py \
16     --config config.yaml \
17     --device gpu \
18     --pool-size 100 \
19     --monitor-interval 300

```

### Garantías de Confiabilidad:

- **Sin OOM Silencioso:** Margen de 30% previene asignaciones inesperadas.
- **Fragmentación Reducida:** Allocador platform libera agresivamente.



- **Uptime Sostenido:** Pool predefinido evita picos de asignación.
- **Degradación Gradual:** Monitoreo detecta saturación tempranamente.

## 6 Persistencia (Atomic Snapshotting)

El sistema implementa persistencia binaria protegida por checksum.

```

1 import hashlib
2 import msgpack
3
4 def save_snapshot(self, filepath: str):
5     """
6     Exporta el estado interno Sigma_t a formato binario (MessagePack).
7     Incluye Checksum SHA-256 al final del archivo.
8     """
9     # Serialización de tensores JAX a bytes
10    state_dict = self._serialize_jax_state(self._state)
11
12    # Segmentación Modular (K-Blocks)
13    # IMPORTANTE: Incluir versionado de schema para evitar errores al cargar
14    # snapshots generados con versiones antiguas (cambios en profundidad de firma, etc.)
15    payload = {
16        "schema_version": self.config.schema_version, # Versionado seguro
17        "timestamp": time.time_ns(),
18        "config": asdict(self.config),
19        "global": state_dict["global"], # rho, G+, ema
20        "telemetry": {
21            "kurtosis": float(self._state.kurtosis),
22            "dgm_entropy": float(self._state.dgm_entropy),
23            "adaptive_threshold": float(self._state.h_adaptive)
24        },
25        "flags": {
26            "degraded_inference": bool(self._state.degraded_mode),
27            "emergency": bool(self._state.emergency_mode),
28            "regime_change": bool(self._state.regime_changed),
29            "mode_collapse": bool(self._state.mode_collapse_warning)
30        },
31        "kernels": {
32            "A": state_dict["kernel_a"],
33            "B": state_dict["kernel_b"],
34            "C": state_dict["kernel_c"],
35            "D": state_dict["kernel_d"]
36        }
37    }
38
39    data_bytes = msgpack.packb(payload)
40    checksum = hashlib.sha256(data_bytes).hexdigest()
41
42    with open(filepath, "wb") as f:
43        f.write(data_bytes)
44        f.write(checksum.encode('utf-8')) # Append hash
45
46 def load_snapshot(self, filepath: str):
47     """
48     Carga estado. Valida SHA-256 y schema_version antes de deserializar.
49     Lanza ValueError si falla la validación o schema incompatible.
50     """
51    with open(filepath, "rb") as f:
52        content = f.read()
53
54    data_bytes = content[:-64] # Todo menos los últimos 64 bytes (SHA256 hex)
55    stored_checksum = content[-64:].decode('utf-8')
56
57    computed = hashlib.sha256(data_bytes).hexdigest()
58    if computed != stored_checksum:
59        raise ValueError("Snapshot corrupto: Checksum mismatch.")
60
61    payload = msgpack.unpackb(data_bytes)
62
63    # Validar schema_version para detectar incompatibilidades
64    loaded_schema = payload.get('schema_version', 'unknown')

```

```

65         if loaded_schema != self.config.schema_version:
66             raise ValueError(
67                 f"Schema version mismatch: snapshot={loaded_schema}, "
68                 f"current={self.config.schema_version}. "
69                 f"Cannot load snapshot generated with incompatible kernel depths or signature
features."
70             )
71
72         self._state = self._deserialize_jax_state(payload)

```

## 7 I/O Asíncrono para Snapshots (Non-Blocking)

**Problema de Latencia:** La función `save_snapshot()` invoca operaciones síncronas de I/O:

1. Serialización `MessagePack` del estado (microsegundos)
2. **Escritura a disco** (milisegundos: 1–100 ms según velocidad de almacenamiento)
3. **Cálculo SHA-256** en los datos (milisegundos: 2–50 ms para estados de 1–100 MB)

Si estas operaciones se ejecutan en el hilo principal del predictor, contaminan el *jitter* de latencia en la predicción:

- El reloj de predicción (ingesta de datos → forward pass de 4 ramas → actualización orquestador) se bloquea.
- En mercados H.F., una desviación de 50 ms es catastrófica (oportunidades perdidas).
- El SLA (Service Level Agreement) de latencia P99 se degrada irremediablemente.

### Solución: Delegación Asíncrona a `ThreadPoolExecutor`

El cálculo de checksum y la escritura a disco se delegan a un *thread pool* dedicado, permitiendo que el hilo principal continúe sin obstáculos:

```

1  import concurrent.futures
2  import hashlib
3  import msgpack
4  import threading
5  import time
6
7  class UniversalPredictor_AsyncIO:
8      def __init__(self, n_worker_threads=2):
9          # Pool de threads dedicados a I/O (no interfieren con inferencia)
10         self.io_executor = concurrent.futures.ThreadPoolExecutor(
11             max_workers=n_worker_threads,
12             thread_name_prefix="snapshot_io_"
13         )
14
15         # Futuro del snapshot en vuelo para monitoreo (opcional)
16         self.pending_snapshot_future = None
17         self.snapshot_lock = threading.Lock()
18
19         def _compute_and_save_async(self, filepath: str, data_bytes: bytes):
20             """
21             Ejecuta en thread pool: calcula checksum y escribe a disco.
22             No bloquea el hilo principal.
23             """
24             checksum = hashlib.sha256(data_bytes).hexdigest()
25
26             # Escritura atómica (write + rename) para evitar estado intermedio
27             temp_filepath = filepath + ".tmp"
28
29             try:
30                 with open(temp_filepath, "wb") as f:
31                     f.write(data_bytes)
32                     f.write(checksum.encode('utf-8'))
33
34                 # Rename atómico (POSIX-compliant)
35                 import os

```

```

36         os.replace(temp_filepath, filepath)
37
38         return {
39             'status': 'success',
40             'filepath': filepath,
41             'filesize_bytes': len(data_bytes),
42             'checksum': checksum,
43             'timestamp': time.time()
44         }
45     except Exception as e:
46         return {
47             'status': 'error',
48             'filepath': filepath,
49             'error': str(e),
50             'timestamp': time.time()
51         }
52
53 def save_snapshot_nonblocking(self, filepath: str) -> concurrent.futures.Future:
54     """
55     Exporta estado a snapshot sin bloquear el hilo de inferencia.
56
57     Retorna un Future<dict>. El llamante puede:
58     - Ignorarlo (fire-and-forget): permitir que escriba en background
59     - Esperar con .result(timeout=N): bloquear solo si es necesario (monitoreo)
60
61     Arquitectura:
62     1. Main thread: Serialización MessagePack (rápido: microsegundos)
63     2. Main thread: Retorna control inmediatamente
64     3. Worker thread: Cálculo SHA-256 + escritura a disco (en background)
65     """
66
67     # Paso 1: Serialización (en el hilo principal, muy rápido)
68     state_dict = self._serialize_jax_state(self._state)
69
70     payload = {
71         "schema_version": self.config.schema_version,
72         "timestamp": time.time_ns(),
73         "config": asdict(self.config),
74         "global": state_dict["global"],
75         "telemetry": {
76             "kurtosis": float(self._state.kurtosis),
77             "dgm_entropy": float(self._state.dgm_entropy),
78             "adaptive_threshold": float(self._state.h_adaptive)
79         },
80         "flags": {
81             "degraded_inference": bool(self._state.degraded_mode),
82             "emergency": bool(self._state.emergency_mode),
83             "regime_change": bool(self._state.regime_changed),
84             "mode_collapse": bool(self._state.mode_collapse_warning)
85         },
86         "kernels": {
87             "A": state_dict["kernel_a"],
88             "B": state_dict["kernel_b"],
89             "C": state_dict["kernel_c"],
90             "D": state_dict["kernel_d"]
91         }
92     }
93
94     data_bytes = msgpack.packb(payload)
95
96     # Paso 2: Delegar I/O a thread pool (no-bloqueante)
97     future = self.io_executor.submit(
98         self._compute_and_save_async,
99         filepath,
100         data_bytes
101     )
102
103     # Mantener referencia al Future para debugging (opcional)
104     with self.snapshot_lock:
105         self.pending_snapshot_future = future
106
107     return future
108

```

```

109 def predict_step_with_async_checkpoint(self, x_t: jnp.ndarray) -> Tuple[jnp.ndarray,
Optional[concurrent.futures.Future]]:
110     """
111     Paso de predicción con snapshot asíncrono periódico.
112
113     El snapshot se dispara cada N pasos pero NO interfiere con latencia de inferencia.
114     """
115     # Predicción principal (hot path)
116     prediction = self._predict_step_core(x_t)
117
118     # Checkpoint asíncrono si es el momento
119     snapshot_future = None
120     if self.step_counter % self.checkpoint_interval == 0:
121         checkpoint_path = f"{self.checkpoint_dir}/snapshot_step_{self.step_counter}.
msgpack"
122         snapshot_future = self.save_snapshot_nonblocking(checkpoint_path)
123         # El hilo retorna SIN esperar a que el snapshot se escriba a disco
124
125     self.step_counter += 1
126     return prediction, snapshot_future
127
128 def monitor_snapshot_queue(self):
129     """
130     Thread de monitoreo (opcional) que reporta el estado de snapshots en vuelo.
131     Ejecutado en otro thread para no interferir.
132     """
133     while not self.shutdown_event.wait(timeout=5.0):
134         with self.snapshot_lock:
135             if self.pending_snapshot_future is not None:
136                 if self.pending_snapshot_future.done():
137                     try:
138                         result = self.pending_snapshot_future.result()
139                         if result['status'] == 'success':
140                             print(f"[INFO] Snapshot guardado: {result['filepath']} "
f"({result['filesize_bytes']} bytes)")
141                         else:
142                             print(f"[WARNING] Snapshot falló: {result['error']}")
143                     except Exception as e:
144                         print(f"[ERROR] Future exception: {e}")
145
146 def graceful_shutdown(self, timeout_seconds=10):
147     """
148     Espera a que todos los snapshots pendientes se completen antes de cerrar.
149     """
150     print("[INFO] Aguardando snapshots pendientes...")
151
152     # Esperar a que se completen (con timeout)
153     concurrent.futures.wait(
154         [self.pending_snapshot_future] if self.pending_snapshot_future else [],
155         timeout=timeout_seconds
156     )
157
158     # Cerrar pool
159     self.io_executor.shutdown(wait=True)
160     print("[INFO] ThreadPoolExecutor cerrado gracefully")
161

```

### Implicaciones de Performance:

Operación	Latencia (ms)	Hilo
Serialización MessagePack	0.1–0.5	Principal (hot path)
SHA-256 en 10 MB	5–15	Worker (background)
Escritura a disco	2–50	Worker (background)
<b>Latencia observed por predictor</b>	<b>0.1–0.5</b>	<b>Principal</b>

Sin I/O asíncrono: latencia observada ~ 7–65 ms Con I/O asíncrono: latencia observada ~ 0.1–0.5 ms **Factor de mejora: 14–130x**

### Garantías Operacionales:

- **Fire-and-Forget:** Ignorar el Future retornado permite que el snapshot se escriba en background sin interferencia.

- **Integridad Atómica:** Escritura a archivo temporal seguida de rename POSIX garantiza que no hay snapshots corruptos parciales.
- **Monitoreo Opcional:** El `thread_monitor_snapshot_queue()` reporta estado sin afectar predicción (ejecutado en thread separado).
- **Graceful Shutdown:** `graceful_shutdown()` aguarda a snapshots pendientes antes de terminar proceso.
- **SLA Garantizado:** Arquitectura no-bloqueante asegura que **P99 latencia** se mantiene  $\leq 1$  ms incluso durante I/O intensivo.

#### Configuración Recomendada:

```

1 predictor = UniversalPredictor_AsyncIO(
2     n_worker_threads=2, # Tipicamente 1-2 threads suficientes
3     checkpoint_interval=1000 # Cada 1000 pasos (~1 segundo en latencia 1ms)
4 )
5
6 # En el loop de trading/predicción:
7 for x_t in market_stream:
8     prediction, snapshot_future = predictor.predict_step_with_async_checkpoint(x_t)
9
10    # USA prediction inmediatamente
11    # El snapshot se escribirá en background
12    if snapshot_future is not None:
13        # Opcional: esperar solo en situaciones críticas (ej. antes de shutdown)
14        snapshot_result = snapshot_future.result(timeout=30)

```

## 8 Apagado Elegante (Graceful Shutdown) para Contenedores

### Problema en Entornos Orquestados (Docker/Kubernetes):

En infraestructura de producción moderna, el predictor opera típicamente en contenedores que pueden ser:

- **Reiniciados:** Por actualizaciones de imagen, cambios de configuración, o healthcheck failures
- **Migrados:** Rebalanceo de carga entre nodos del cluster (rolling updates, node draining)
- **Escalados:** Auto-scaling que crea/destruye pods según demanda
- **Terminados:** Reclamación de recursos o fallos de infraestructura

En todos estos casos, Kubernetes/Docker envía **SIGTERM** (señal POSIX 15) al proceso principal, dándole *grace period* (típicamente 30 segundos) antes de enviar **SIGKILL** (forzado, no capturado).

### Consecuencia de Apagado Abrupto (sin manejo de SIGTERM):

1. **Pérdida de Inercia del Orquestador:** Los pesos  $\rho_i$  entrenados se pierden si no están en snapshot reciente
2. **Corrupción de Buffer SIA:** El buffer circular de WTMM queda truncado, perdiendo contexto histórico
3. **Estado CUSUM Inconsistente:** Las variables  $C^+$ ,  $C^-$  no reflejan el régimen real al reiniciar
4. **Snapshots Parciales:** Si un `save_snapshot_nonblocking()` estaba en vuelo, el archivo puede corromperse
5. **Datos en Buffer Residual:** Observaciones recibidas pero no procesadas se descartan

Resultado: El sistema reinicia en *estado frío*, perdiendo hasta 30 minutos de aprendizaje adaptativo (dependiendo del intervalo de snapshots).

### Solución: Captura de SIGTERM con Protocolo de Shutdown Ordenado

Implementar un manejador de señales POSIX que intercepte SIGTERM, bloquee nuevas entradas, procese el backlog, persista estado final, y termine limpiamente con código de salida 0.

### Implementación:

```

1 import signal
2 import sys
3 import threading
4 import time
5 import logging
6 from typing import Optional
7
8 class UniversalPredictor_GracefulShutdown:
9     def __init__(self, config: PredictorConfig):
10         self.config = config
11         self.predictor = UniversalPredictor_AsyncIO(config)
12
13         # Control de ciclo de vida
14         self.shutdown_requested = threading.Event()
15         self.is_accepting_data = True
16         self.input_buffer_lock = threading.Lock()
17         self.residual_buffer = [] # Buffer de observaciones no procesadas
18
19         # Registro de señales POSIX
20         signal.signal(signal.SIGTERM, self._handle_sigterm)
21         signal.signal(signal.SIGINT, self._handle_sigint) # Ctrl+C en desarrollo
22
23         # Logger para auditoría
24         self.logger = logging.getLogger("predictor.shutdown")
25         self.logger.info("[INIT] Graceful shutdown handler registered")
26
27     def _handle_sigterm(self, signum, frame):
28         """
29         Manejador de SIGTERM (enviado por Kubernetes/Docker).
30         Inicia protocolo de apagado elegante.
31         """
32         self.logger.warning(f"[SIGTERM] Received signal {signum}. Initiating graceful shutdown
33         ...")
34         self.shutdown_requested.set()
35
36     def _handle_sigint(self, signum, frame):
37         """
38         Manejador de SIGINT (Ctrl+C local, desarrollo).
39         """
40         self.logger.warning(f"[SIGINT] Received signal {signum}. Initiating graceful shutdown
41         ...")
42         self.shutdown_requested.set()
43
44     def accept_observation(self, obs: MarketObservation) -> Optional[PredictionResult]:
45         """
46         Método thread-safe para recibir observaciones.
47         Rechaza nuevas entradas si shutdown está en curso.
48         """
49         if self.shutdown_requested.is_set() or not self.is_accepting_data:
50             self.logger.warning(f"[REJECT] Observation rejected (shutdown in progress): {obs.
51             timestamp_ns}")
52             return None
53
54         with self.input_buffer_lock:
55             self.residual_buffer.append(obs)
56
57         # Procesamiento inmediato (o delegación a cola)
58         return self._process_observation(obs)
59
60     def _process_observation(self, obs: MarketObservation) -> PredictionResult:
61         """
62         Procesa observación y emite predicción.
63         """
64         result = self.predictor.predict_next(obs)
65
66         with self.input_buffer_lock:
67             # Remover de buffer residual ya que se procesó
68             if obs in self.residual_buffer:
69                 self.residual_buffer.remove(obs)
70
71         return result
72
73     def graceful_shutdown(self, timeout_seconds: int = 25):

```

```

71     """
72     Protocolo de apagado elegante (llamado automáticamente por SIGTERM).
73
74     Pasos:
75     1. Bloquear nuevas observaciones
76     2. Procesar buffer residual
77     3. Esperar snapshots asíncronos pendientes
78     4. Ejecutar snapshot final
79     5. Cerrar recursos (sockets, threads)
80     6. Salir con código 0
81
82     Args:
83         timeout_seconds: Tiempo máximo antes de rendirse (debe ser < grace period de K8s)
84     """
85     shutdown_start = time.time()
86     self.logger.info("                ")
87     self.logger.info("    GRACEFUL SHUTDOWN INITIATED")
88     self.logger.info("                ")
89
90     # PASO 1: Bloquear nuevas entradas
91     self.logger.info("[1/5] Blocking new observations...")
92     self.is_accepting_data = False
93     time.sleep(0.1) # Dar tiempo a threads en vuelo para terminar
94     self.logger.info("                Input blocked: ")
95
96     # PASO 2: Procesar buffer residual
97     self.logger.info("[2/5] Processing residual buffer...")
98     with self.input_buffer_lock:
99         residual_count = len(self.residual_buffer)
100         self.logger.info(f"                Residual observations: {residual_count}")
101
102         for obs in list(self.residual_buffer):
103             if time.time() - shutdown_start > timeout_seconds - 10:
104                 self.logger.warning("                Timeout approaching, aborting residual
processing")
105                 break
106
107             try:
108                 _ = self._process_observation(obs)
109             except Exception as e:
110                 self.logger.error(f"                Error processing residual: {e}")
111
112         remaining = len(self.residual_buffer)
113         if remaining > 0:
114             self.logger.warning(f"                {remaining} observations dropped (timeout)")
115         else:
116             self.logger.info("                Residual buffer cleared: ")
117
118     # PASO 3: Esperar snapshots asíncronos pendientes
119     self.logger.info("[3/5] Waiting for pending async snapshots...")
120     pending_snapshot = self.predictor.pending_snapshot_future
121
122     if pending_snapshot is not None and not pending_snapshot.done():
123         try:
124             remaining_time = max(1, timeout_seconds - (time.time() - shutdown_start))
125             pending_snapshot.result(timeout=remaining_time)
126             self.logger.info("                Async snapshot completed: ")
127         except concurrent.futures.TimeoutError:
128             self.logger.error("                Async snapshot timed out (may be corrupted)")
129         except Exception as e:
130             self.logger.error(f"                Async snapshot failed: {e}")
131     else:
132         self.logger.info("                No pending snapshots: ")
133
134     # PASO 4: Snapshot final (síncrono, garantizado)
135     self.logger.info("[4/5] Executing final snapshot (synchronous)...")
136     try:
137         final_snapshot_path = f"snapshots/shutdown_{int(time.time())}.pkl"
138         self.predictor.save_snapshot(final_snapshot_path)
139         self.logger.info(f"                Final snapshot saved: {final_snapshot_path}")
140     except Exception as e:
141         self.logger.error(f"                Final snapshot FAILED: {e}")
142     # Continuar de todos modos, no queremos bloquear el shutdown

```

```

143
144     # PASO 5: Cerrar recursos
145     self.logger.info("[5/5] Closing resources...")
146     try:
147         # Cerrar ThreadPoolExecutor (si existe)
148         if hasattr(self.predictor, 'io_executor'):
149             self.predictor.io_executor.shutdown(wait=True, cancel_futures=False)
150             self.logger.info("        ThreadPoolExecutor closed: ")
151
152         # Cerrar sockets de mercado (si aplica)
153         # self.market_socket.close()
154
155         # Limpiar caché JAX (opcional)
156         # jax.clear_caches()
157
158     except Exception as e:
159         self.logger.error(f"        Error closing resources: {e}")
160
161     # Resumen final
162     total_time = time.time() - shutdown_start
163     self.logger.info("        ")
164     self.logger.info(f"    SHUTDOWN COMPLETED ({total_time:.2f}s)")
165     self.logger.info("        ")
166
167     # Salir con código 0 (limpio)
168     sys.exit(0)
169
170     def run_forever(self):
171         """
172         Bucle principal del predictor.
173         Monitorea shutdown_requested en cada iteración.
174         """
175         self.logger.info("[MAIN] Predictor running. Waiting for market data...")
176
177         try:
178             while not self.shutdown_requested.is_set():
179                 # Simular recepción de datos (en producción: socket.recv, kafka.poll, etc.)
180                 # observation = self.market_socket.recv()
181                 # result = self.accept_observation(observation)
182
183                 time.sleep(0.001) # Evitar busy-wait
184
185         except KeyboardInterrupt:
186             # Capturado por _handle_sigint, pero por si acaso
187             self.logger.warning("[MAIN] KeyboardInterrupt detected")
188
189         # Shutdown solicitado
190         self.graceful_shutdown(timeout_seconds=25)

```

## Integración con Kubernetes ConfigMap:

```

1 # deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: universal-predictor
6 spec:
7   replicas: 3
8   template:
9     spec:
10       terminationGracePeriodSeconds: 30 # Tiempo para graceful_shutdown()
11       containers:
12       - name: predictor
13         image: predictor:v1.0
14         command: ["python", "main.py"]
15         lifecycle:
16           preStop:
17             exec:
18               # Kubernetes ejecuta esto ANTES de enviar SIGTERM
19               # Opcional: Notificar load balancer que el pod está muriendo
20               command: ["/bin/sh", "-c", "sleep 2"]
21       resources:
22         requests:
23           memory: "16Gi"

```



```

24         nvidia.com/gpu: 1
25     limits:
26         memory: "32Gi"
27         nvidia.com/gpu: 1

```

### Script Principal (main.py):

```

1  #!/usr/bin/env python3
2  import logging
3  from stochastic_predictor import UniversalPredictor_GracefulShutdown, PredictorConfig
4
5  def main():
6      # Configurar logging
7      logging.basicConfig(
8          level=logging.INFO,
9          format='%(asctime)s [%(name)s] %(levelname)s: %(message)s',
10         handlers=[
11             logging.StreamHandler(),
12             logging.FileHandler("/var/log/predictor.log")
13         ]
14     )
15
16     logger = logging.getLogger("main")
17     logger.info("Starting Universal Predictor...")
18
19     # Cargar configuración
20     config = PredictorConfig.from_json("config.json")
21
22     # Inicializar predictor con graceful shutdown
23     predictor = UniversalPredictor_GracefulShutdown(config)
24
25     # Ejecutar (bloqueante hasta SIGTERM)
26     predictor.run_forever()
27
28     # NUNCA llega aquí (sys.exit(0) en graceful_shutdown)
29
30 if __name__ == "__main__":
31     main()

```

### Prueba de Apagado Elegante:

```

1  def test_graceful_shutdown_protocol():
2      """
3      Simula apagado elegante con buffer residual y snapshot pendiente.
4      """
5      import signal
6      import os
7      import time
8
9      config = PredictorConfig(checkpoint_interval=100)
10     predictor = UniversalPredictor_GracefulShutdown(config)
11
12     # Simular carga de trabajo
13     for i in range(50):
14         obs = MarketObservation(
15             price=jnp.array([float(i)]),
16             target=jnp.array([0.0]),
17             timestamp_ns=time.time_ns()
18         )
19         predictor.accept_observation(obs)
20
21     # Disparar snapshot asíncrono
22     predictor.predictor.save_snapshot_nonblocking("test_snapshot.pkl")
23
24     # Añadir observaciones al buffer residual
25     for i in range(10):
26         obs = MarketObservation(
27             price=jnp.array([float(100 + i)]),
28             target=jnp.array([0.0]),
29             timestamp_ns=time.time_ns()
30         )
31         with predictor.input_buffer_lock:
32             predictor.residual_buffer.append(obs)
33
34     # Simular SIGTERM (en test, llamamos directamente al handler)

```

```

35 predictor._handle_sigterm(signal.SIGTERM, None)
36
37 # Verificar que shutdown se ejecutó
38 assert predictor.shutdown_requested.is_set(), "Shutdown not triggered"
39
40 # Ejecutar shutdown manualmente (en producción es automático)
41 try:
42     predictor.graceful_shutdown(timeout_seconds=10)
43 except SystemExit as e:
44     assert e.code == 0, f"Exit code should be 0, got {e.code}"
45
46 # Verificar que snapshot final existe
47 final_snapshots = [f for f in os.listdir("snapshots") if f.startswith("shutdown_")]
48 assert len(final_snapshots) > 0, "Final snapshot not created"
49
50 print(" [] Graceful shutdown test PASSED")
51 print(f"    Final snapshot: {final_snapshots[-1]}")
52 print(f"    Residual buffer: cleared")
53 print(f"    Async snapshots: completed")

```

### Cronología de Shutdown (Típica):

Tiempo (s)	Evento	Acción
$t = 0.000$	Kubernetes: <code>kubectl delete pod predictor-xyz</code>	Envía SIGTERM al PID 1
$t = 0.001$	Python: <code>_handle_sigterm()</code> captura señal	Set <code>shutdown_requested</code>
$t = 0.002$	Predictor: Bloquea nuevas observaciones	<code>is_accepting_data = False</code>
$t = 0.010$	Predictor: Procesa buffer residual (10 obs)	<code>Loop_process_observation()</code>
$t = 0.500$	Predictor: Espera snapshot asíncrono en vuelo	<code>future.result(timeout=20)</code>
$t = 1.200$	Predictor: Snapshot final (síncrono)	<code>save_snapshot()</code>
$t = 1.800$	Predictor: Cierra <code>ThreadPoolExecutor</code>	<code>executor.shutdown(wait=True)</code>
$t = 2.000$	Python: <code>sys.exit(0)</code>	Proceso termina limpiamente
$t = 2.001$	Kubernetes: Recibe exit code 0	Marca pod como <code>Completed</code>

### Comparación: Shutdown Abrupto vs Elegante

Métrica	Abrupto (SIGKILL)	Elegante (SIGTERM)
Pérdida de datos en buffer	Sí (100%)	No (procesados)
Estado del Orquestador	Perdido (si sin snapshot)	Guardado (snapshot final)
Snapshots asíncronos	Corruptos	Completados
Buffer SIA/WTMM	Truncado	Persistido
Estado CUSUM	Inconsistente	Sincronizado
Tiempo de recuperación	30+ min (cold start)	<1 min (hot reload)
Integridad de datos	Riesgo alto	Garantizada
Exit code	137 (killed)	0 (success)

### Configuración de Timeouts (Recomendaciones):

- **Kubernetes `terminationGracePeriodSeconds`:** 30 segundos (estándar)
- **Python `graceful_shutdown(timeout_seconds)`:** 25 segundos (margen de 5s)
- **Snapshot asíncrono `result(timeout)`:** 20 segundos (dentro del timeout de shutdown)
- **Buffer residual processing:** Máximo 10 segundos o 1000 observaciones

### Consideraciones de Producción:

1. **Snapshots Incrementales:** Si el estado es muy grande (>100 MB), considerar snapshots diferenciales para reducir tiempo de escritura
2. **Health Probes:** Kubernetes debe tener `readinessProbe` que falle inmediatamente cuando `is_accepting_data = False`, para que el load balancer deje de enviar tráfico
3. **Pre-Stop Hook:** Usar `lifecycle.preStop` para notificar a sistemas externos (ej. API Gateway) que el pod está en shutdown

4. **Persistencia Distribuida:** Guardar snapshots finales en almacenamiento compartido (NFS, S3) en lugar de volumen local del pod
5. **Logging Estructurado:** Todos los logs de shutdown deben ser JSON con campo `shutdown_phase` para análisis post-mortem
6. **Métricas de Shutdown:** Exponer contador Prometheus `predictor_graceful_shutdowns_total` y histogram `predictor_shutdown_duration_seconds`

#### Integración con Prometheus:

```

1 from prometheus_client import Counter, Histogram
2
3 class UniversalPredictor_GracefulShutdown_Monitored:
4     def __init__(self, config: PredictorConfig):
5         # ... (inicialización base) ...
6
7         # Métricas de shutdown
8         self.shutdown_counter = Counter(
9             'predictor_graceful_shutdowns_total',
10            'Total number of graceful shutdowns executed'
11        )
12        self.shutdown_duration = Histogram(
13            'predictor_shutdown_duration_seconds',
14            'Time taken to complete graceful shutdown',
15            buckets=[0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0]
16        )
17        self.residual_observations = Histogram(
18            'predictor_shutdown_residual_observations',
19            'Number of observations in buffer during shutdown',
20            buckets=[0, 1, 5, 10, 50, 100, 500, 1000]
21        )
22
23    def graceful_shutdown(self, timeout_seconds: int = 25):
24        self.shutdown_counter.inc()
25
26        with self.shutdown_duration.time():
27            # ... (código de shutdown) ...
28
29            with self.input_buffer_lock:
30                self.residual_observations.observe(len(self.residual_buffer))
31
32            # ... (resto del shutdown) ...

```

#### Conclusión:

El **Graceful Shutdown** es *obligatorio* para cualquier sistema stateful desplegado en contenedores. Sin captura de SIGTERM:

- Rolling updates de Kubernetes destruyen el estado entrenado del Orquestador
- Node draining causa pérdida de 30+ minutos de adaptación a régimen actual
- Auto-scaling down elimina snapshots en vuelo, corrompiendo persistencia
- Exit code 137 (SIGKILL) indica fallo en logs, complicando debugging

Con el protocolo implementado:

- 0% pérdida de datos (buffer residual procesado)
- Estado completo persistido en <2 segundos
- Recuperación en <1 minuto (vs 30+ minutos cold start)
- Exit code 0 → logs limpios, métricas de salud correctas
- Compatible con rolling updates sin downtime de predicción

El overhead de implementación es *mínimo* (100 líneas de código) comparado con el *riesgo operacional masivo* de no implementarlo.

## 9 Ajuste Adaptativo del Umbral CUSUM

El sistema implementa el **Lema de Umbral Adaptativo** basado en curtosis, permitiendo que el detector CUSUM se ajuste automáticamente a regímenes con colas pesadas.

### 9.1 Fórmula de Ajuste

El umbral de detección de cambio de régimen se calcula dinámicamente:

$$h_t = k \cdot \sigma_t \cdot \left(1 + \ln\left(\frac{\kappa_t}{3}\right)\right)$$

donde:

- $k$ : Slack calibrado (`cusum_k` en configuración)
- $\sigma_t$ : Volatilidad EMA del error de predicción
- $\kappa_t$ : Curtosis empírica móvil (ventana de 252 pasos)
- 3: Curtosis de referencia Gaussiana

### 9.2 Interpretación de Curtosis

Rango $\kappa_t$	Régimen de Mercado
$\kappa_t \approx 3$	Gaussiano (mercado normal)
$\kappa_t \in [5, 10]$	Volatilidad financiera estándar
$\kappa_t \in [10, 15]$	Alta volatilidad (eventos outlier)
$\kappa_t > 15$	Régimen de crisis (colas pesadas)
$\kappa_t > 20$	Falla en modelo de residuos (alerta crítica)

Table 1: Interpretación de curtosis empírica

**Nota:** El ajuste logarítmico permite que el umbral se expanda automáticamente cuando  $\kappa_t > 3$ , evitando falsos positivos en regímenes de alta curtosis mientras mantiene sensibilidad a cambios estructurales genuinos.

## 10 Periodo de Gracia (Ventana Refractaria) Post-Cambio de Régimen

### 10.1 Motivación

Cuando CUSUM detecta un cambio de régimen ( $G^+ > h_t$ ), el orquestador reinicia los pesos a distribución uniforme y reseta los acumuladores CUSUM. Sin embargo, en los pasos inmediatos posteriores:

- **Volatilidad inflada:** El error de predicción  $e_t$  se vuelve temporalmente grande porque los nuevos pesos uniformes aún no se han optimizado.
- **Curtosis elevada:** El buffer de errores refuerza momentáneamente momentos de alto orden.
- **Cascada de falsas alarmas:** CUSUM podría detectar "otro cambio" basándose en ruido de recalibración, no en genuina ruptura estructural.

Esto puede causar oscilación patológica donde el sistema alterna entre reinicio uniforme y redetección espuria.

## 10.2 Solución: Grace Period (Refractario)

Se introduce un parámetro `grace_period_steps` en `PredictorConfig` (por defecto 20-50 pasos):

$$\text{CUSUM\_silenciado} = (\text{pasos\_desde\_último\_cambio} < \text{grace\_period\_steps})$$

**Durante el período de gracia:**

1. El detector CUSUM calcula su estadística  $G^+$  internamente (para diagnóstico)
2. **Pero no emite alarma** (`regime_change_detected = False`) aunque  $G^+ > h_t$
3. El acumulador  $G^+$  se mantiene en reset ( $G^+ = 0$  al inicio del glance)
4. Permiten que los pesos converjan bajo el algoritmo JKO sin interrupciones

**Transcurrido el período:**

- CUSUM vuelve a estado operacional normal
- Próxima detección de cambio (si ocurre) desencadena nuevo período de gracia

## 10.3 Algoritmo de Implementación

```
1 class CUSUMState:
2     def __init__(self, grace_period_steps=20):
3         self.g_plus = 0.0
4         self.g_minus = 0.0
5         self.error_sq_ema = 0.0
6         self.steps_since_regime_change = 0
7         self.grace_period = grace_period_steps
8
9     def step(self, error, sigma_t, kurtosis):
10        """Avanza el estado CUSUM con silenciamiento refractario."""
11        # Incrementar contador desde último cambio
12        self.steps_since_regime_change += 1
13
14        # Calcular estadística (siempre)
15        k = self.config.cusum_k
16        h_adaptive = k * sigma_t * (1 + np.log(max(kurtosis, 1.0) / 3.0))
17
18        s_standardized = np.abs(error) / sigma_t
19        s_centered = s_standardized - 1.0
20
21        self.g_plus = max(0.0, self.g_plus + s_centered - k)
22
23        # Lógica de alarma CON GRACIA
24        is_in_grace_period = (
25            self.steps_since_regime_change < self.grace_period
26        )
27
28        if is_in_grace_period:
29            # Silenciar: no emitir alarma
30            alarm = False
31        else:
32            # Normal: comparar con umbral
33            alarm = (self.g_plus > h_adaptive)
34
35        return alarm, self.g_plus, h_adaptive
36
37    def reset_on_regime_change(self):
38        """Al detectar cambio, iniciar período de gracia."""
39        self.g_plus = 0.0
40        self.steps_since_regime_change = 0 # Reiniciar reloj
```

grace_period_steps	Escenario	Justificación
10-15	Mercados estables, baja latencia	Recalibración rápida
20-30	Mercados con volatilidad media	Balance entre estabilidad y reactividad
40-50	Mercados de alta turbulencia	Mayor tiempo para convergencia JKO
60+	Instrumentos ilíquidos o con gaps	Minimizar oscilaciones patológicas

Table 2: Recomendaciones para grace\_period\_steps según régimen

## 10.4 Parámetros Sugeridos

## 10.5 Diagnóstico y Telemetría

Se recomienda registrar (sin decidir) durante el período de gracia:

- $G^+$  observable (si hubiera alarma)
- $\sigma_t$  y  $\kappa_t$  instantáneos
- Convergencia del JKO (distancia Wasserstein a cada paso)

Esto permite post-hoc análisis de si el período fue suficiente o excesivo.

# 11 Flags de Operación y Recuperación

El sistema mantiene cuatro flags booleanos explícitos que señalizan estados críticos al ejecutor:

## 11.1 DegradedInferenceMode

**Condición de activación:**

$$\text{TTL}(y_{\text{target}}) = t_{\text{current}} - t_{\text{signal}} > \Delta_{\text{max}}$$

**Implicaciones operacionales:**

1. Suspende actualización del transporte JKO inmediatamente
2. Congela pesos  $\rho$  en último valor válido (modo inercial)
3. Predicciones continúan generándose pero con confianza degradada
4. Riesgo NO está siendo optimizado geométricamente

**Recuperación con histéresis:**

$$\text{TTL}(y_{\text{target}}) < h_{\text{hyst}} \cdot \Delta_{\text{max}}$$

donde  $h_{\text{hyst}} = \text{inference\_recovery\_hysteresis}$  (por defecto 0.8) parametrizable en PredictorConfig.

Se emite NormalOperationRestoredEvent al recuperar.

## 11.2 EmergencyMode

**Condición:**  $H_t < H_{\text{min}}$  (singularidad crítica detectada)

**Acción:** Fuerza  $w_D \rightarrow 1.0$  (Kernel D de signatures) y cambia a métrica de Huber robusta.

## 11.3 RegimeChangeDetected

**Condición:**  $G^+ > h_t$  (CUSUM detecta cambio de régimen)

**Acción:** Reinicio de entropía a distribución uniforme y reset de acumuladores.

## 11.4 ModeCollapseWarning

**Condición:**  $H_{\text{DGM}} < \gamma \cdot H[g]$  durante  $> 10$  pasos consecutivos (solo relevante si  $\rho_B > 0.05$ )

**Acción correctiva:** Reducir  $\rho_B \rightarrow 0$  hasta re-entrenar red DGM.

## 11.5 Período de Gracia (Refractario) Post-Cambio de Régimen

**Motivación:** Cuando CUSUM detecta un cambio de régimen y resetea los pesos a distribución uniforme, la curtosis y varianza residual se vuelven **transitoriamente infladas** mientras los filtros (SIA, WTMM, EMA) se recalibran. Sin protección, esto provoca **cascadas de falsos positivos** inmediatos: el sistema detecta el mismo cambio repetidamente en los siguientes 5-10 pasos.

**Solución:** Introducir un contador refractario que **silencia CUSUM temporalmente** tras la detección de cambio.

### 11.5.1 Implementación

Dentro del estado interno del predictor, se mantiene un contador:

```
1 @dataclass
2 class PredictorState:
3     # ... otros campos ...
4     grace_period_counter: int = 0 # Contador refractario (decrementado cada paso)
5     regime_change_locked: bool = False # Flag de bloqueo durante gracia
```

La lógica en el núcleo de actualización (dentro de `_core_update_step`) es:

```
1 def _core_update_step(state, price, target, freeze_weights=False):
2     # ... paso de identificación (SIA, WTMM) ...
3
4     # Calcular CUSUM normalmente
5     raw_alarm = self._check_regime_change_with_kurtosis(error)
6
7     # Aplicar período de gracia: silenciar falsa alarma si dentro de gracia
8     if state.grace_period_counter > 0:
9         raw_alarm = False # Suprimir detección durante refractario
10        state.grace_period_counter -= 1
11        state.regime_change_locked = True
12    else:
13        state.regime_change_locked = False
14
15    # Si se detecta cambio FUERA del período de gracia, resetear contador
16    if raw_alarm and not state.regime_change_locked:
17        state.grace_period_counter = self.config.grace_period_steps
18        # Reiniciar pesos a uniforme, resetear acumuladores
19        weights = jnp.ones(4) / 4.0
20        cusum_state['g_plus'] = 0.0
21
22    # ... resto de la lógica ...
23    return new_state, result
```

### 11.5.2 Dinámica Temporal

**Ejemplo:** Con `grace_period_steps=20`:

Paso	CUSUM Crudo	Contador Gracia	Alarma Emitida
$t = 0$	False	0	False
$t = 1$	True	0	True $\Rightarrow$ Cambio detectado
$t = 2$	True	19	False (silenciado)
$t = 3$	True	18	False (silenciado)
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$t = 20$	True	1	False (silenciado)
$t = 21$	True	0	True $\Rightarrow$ Nueva alarma (fin gracia)

Table 3: Evolución del contador de gracia con detecciones repetidas

#### Interpretación:

1. En  $t = 1$ : Cambio genuino detectado. Se resetean pesos, inicia contador `grace_period_counter=20`.
2. En  $t \in [2, 20]$ : Aunque CUSUM crudo sigue alto (inflación transitoria), la alarma es **suprimida**. El sistema se recalibra en silencio.

3. En  $t = 21$  (fin del período): Si la volatilidad persiste (e.g., un verdadero régimen de crisis), la alarma se re-emite. Si fue transitorio, CUSUM se normaliza.

### 11.5.3 Parámetros Recomendados

- **grace\_period\_steps**: Típicamente 10 – 50 pasos.
  - **10-15 pasos**: Para mercados de alta frecuencia ( $> 1$  kHz). Recalibración rápida.
  - **20-30 pasos**: Para operaciones intraday (1-100 Hz). Balance entre rechazo de ruido y reacción.
  - **40-50 pasos**: Para datos de baja frecuencia ( $< 1$  Hz). Permite amortiguamiento completo de transiente.

Entre mercados: Calibrar via `optuna.optimize()` sobre ventanas de validación histórica, minimizando tasa de falsas alarmas dentro de 30 minutos post-cambio.

### 11.5.4 Ventajas

- **Evita Cascadas**: Una sola alarma genuina no dispara falsos positivos en cadena.
- **Permite Recalibración**: Los filtros (SIA, curtosis, varianza) tienen tiempo para estabilizarse sin distorsiones retroactivas.
- **Preserva Reactividad**: Tras el período de gracia, el sistema es tan reactivo como antes.
- **Parametrizable**: Fácil sintonización per-activo o per-mercado.

## 12 Manejo de Errores y Excepciones

### 12.1 Excepciones Estándar

- **DomainError**: Se lanza (o se loguea crítico) si  $y_t$  excede los límites (Outlier Catastrófico  $> 20\sigma$ ).
- **StalenessWarning**: Emitido mediante el sistema de logging estándar de Python cuando se activa la protección TTL.
- **FrequencyWarning**: Alerta si la tasa de arribo de eventos cae por debajo del límite de Nyquist para el análisis de Besov.
- **IntegrityError**: Fallo crítico en la carga de snapshot. El sistema debe abortar y solicitar reinicio en frío.

### 12.2 Alertas Específicas Avanzadas

- **ModeDegradationAlert**: Se emite cuando  $H_{\text{DGM}}$  viola umbral durante  $> 10$  pasos consecutivos. Indica colapso de modo en el predictor neuronal DGM (Rama B).
- **KurtosisOutlierWarning**: Se emite si  $\kappa_t > 20$  de forma persistente ( $> 5$  pasos consecutivos). Señala falla potencial en el modelo de residuos y sugiere revisión de arquitectura.
- **NormalOperationRestoredEvent**: Se emite al recuperar de **DegradedInferenceMode** (cuando TTL vuelve bajo el umbral con histéresis). Señaliza al ejecutor que puede retomar operación normal.



## 12.3 Ejemplo de Logging en Producción

```
1 import logging
2 import os
3 from datetime import datetime
4
5 def save_emergency_dump(predictor, result, asset_id: str):
6     """
7     Guarda un "Dump de Depuración" completo cuando se activa EmergencyMode.
8     Incluye: estado de pesos, buffer de señales, historial de telemetría.
9     """
10    dump_dir = os.path.expanduser("~/predictor_emergency_dumps")
11    os.makedirs(dump_dir, exist_ok=True)
12
13    timestamp = datetime.now().isoformat()
14    dump_file = f"{dump_dir}/{asset_id}_emergency_{timestamp}.msgpack"
15
16    debug_payload = {
17        "emergency_timestamp": timestamp,
18        "asset_id": asset_id,
19        "holder_exponent": float(result.holder_exponent),
20        "weights": [float(w) for w in result.weights],
21        "signal_buffer": predictor._state.signal_circular_buffer.tolist(),
22        "regime_history": predictor._state.cusum_history.tolist(),
23        "telemetry_snapshot": {
24            "kurtosis": float(result.kurtosis),
25            "dgm_entropy": float(result.dgm_entropy),
26            "adaptive_threshold": float(result.adaptive_threshold),
27            "distance_to_collapse": float(result.distance_to_collapse)
28        },
29        "flags_at_emergency": {
30            "degraded_inference": bool(result.degraded_inference_mode),
31            "regime_change": bool(result.regime_change_detected),
32            "mode_collapse": bool(result.mode_collapse_warning)
33        }
34    }
35
36    with open(dump_file, "wb") as f:
37        msgpack.packb(debug_payload, file=f)
38
39    logging.critical(f"Emergency dump saved to {dump_file} for forensius analysis")
40
41 def process_prediction(predictor, obs):
42    result = predictor.step(obs)
43    asset_id = obs.asset_id if hasattr(obs, 'asset_id') else "unknown"
44
45    # Flags críticos
46    if result.degraded_inference_mode:
47        logging.warning(
48            "DEGRADED MODE: TTL exceeded. Weights frozen. "
49            "Consider reducing position size."
50        )
51
52    if result.emergency_mode:
53        logging.critical(
54            f"EMERGENCY: Singularity detected (H={result.holder_exponent:.3f}). "
55            "Forcing Kernel D with Huber loss."
56        )
57    # Guardar dump automáticamente para análisis post-mortem
58    save_emergency_dump(predictor, result, asset_id)
59
60    if result.mode_collapse_warning:
61        logging.error(
62            f"MODE COLLAPSE: DGM entropy below threshold. "
63            f"H_DGM = {result.dgm_entropy:.3f}. "
64            "Reducing rho_B -> 0."
65        )
66
67    if result.kurtosis > 20.0:
68        logging.warning(
69            f"KURTOSIS OUTLIER: kappa = {result.kurtosis:.2f} > 20. "
70            "Residual model may be invalid."
71        )
```

```

72
73 return result

```

## 13 Detección de Mode Collapse en DGM

El sistema monitoriza la entropía diferencial del predictor neuronal (Rama B) para detectar colapso a soluciones triviales.

### 13.1 Criterio de Detección

La entropía diferencial de la solución DGM  $V_\theta(x, t)$  se calcula como:

$$H_{\text{DGM}} = - \int p_V(v) \log p_V(v) dv$$

se compara contra la entropía de la condición terminal  $H[g]$ :

$$H_{\text{DGM}} \geq \gamma \cdot H[g], \quad \gamma \in [0.5, 1.0]$$

Si la violación persiste durante  $> 10$  pasos consecutivos, se activa `mode_collapse_warning`.

### 13.2 Acción Correctiva

El orquestador JKO debe reducir el peso de la Rama B:

$$\rho_B \rightarrow 0$$

hasta que se re-entrene la red neuronal DGM con hiperparámetros ajustados (tasa de aprendizaje, arquitectura, inicialización).

**Nota Teórica:** Una solución colapsada tiene  $H[V_\theta] \rightarrow -\infty$  (distribución delta), correspondiendo a una política de control degenerada que no responde a variaciones del estado.

## 14 Determinismo de Punto Flotante (Bit-Exact Reproducibility)

**Problema:** Para validar la consistencia del sistema en **tests de portabilidad** (ejecución en CPU vs GPU vs FPGA, descritos en *Pruebas.tex*), los cálculos de punto flotante deben ser estrictamente deterministas. Sin embargo, JAX compila a código XLA que puede reordenar operaciones de reducción (ej. `jax.numpy.sum`, llamadas en el algoritmo de Sinkhorn) dependiendo del backend.

**Impacto:**

- En CPU: suma secuencial  $\rightarrow$  error de redondeo  $\epsilon_{\text{CPU}}$
- En GPU: suma paralela con diferentes agrupaciones  $\rightarrow$  error  $\epsilon_{\text{GPU}} \neq \epsilon_{\text{CPU}}$
- En FPGA: precisión de punto flotante custom  $\rightarrow$  error  $\epsilon_{\text{FPGA}}$

Aunque todos los valores sean matemáticamente correctos (dentro de tolerancia numérica), el *bit-exact* resultado difiere, rompiendo tests determinísticos de regresión.

**Solución: Configuración Determinista de XLA y PRNG Fijo**

Fijar las variables de entorno antes de importar JAX:

```

1 import os
2 import jax
3 import jax.numpy as jnp
4
5 # PASO 1: Configurar variables de entorno ANTES de cualquier operación JAX
6 os.environ['XLA_FLAGS'] = '--xla_cpu_use_cross_replica_callbacks=false'
7 os.environ['JAX_DETERMINISTIC_REDUCTIONS'] = '1' # Force deterministic reductions
8 os.environ['JAX_TRACEBACK_FILTERING'] = 'off' # Completo traceback para debugging
9
10 # Alternativas según backend:
11 # Para GPU (CUDA):

```

```

12 # os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
13 # os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':16:8' # Fijo cuBLAS workspace
14
15 # Para forzar CPU (deshabilitar GPU):
16 # os.environ['JAX_PLATFORMS'] = 'cpu'
17
18 # PASO 2: Fijar semillas globales de PRNG
19 import numpy as np
20
21 RANDOM_SEED = 42 # Determinista en todos los backends
22
23 # Semilla NumPy
24 np.random.seed(RANDOM_SEED)
25
26 # Semilla JAX (RNG de clave)
27 from jax import random
28 jax.config.update('jax_default_prng_impl', 'threefry2x32') # Determinista entre backends
29 key = random.PRNGKey(RANDOM_SEED)
30
31 # PASO 3: Importar y configurar JAX después de las variables de entorno
32 jax.config.update('jax_enable_x64', True) # float64 para mayor precisión
33
34 # PASO 4: Verificar que se forzó determinismo
35 print(f"XLA Backend: {jax.devices()}")
36 print(f"JAX Deterministic Mode: {os.environ.get('JAX_DETERMINISTIC_REDUCTIONS', 'not set')}")

```

### Implicaciones Matemáticas:

Consideramos una operación de reducción típica en Sinkhorn (divergencia Kullback-Leibler):

$$D_{\text{KL}}(p||q) = \sum_{i=1}^n p_i \log \left( \frac{p_i}{q_i} \right)$$

En punto flotante, el orden de suma influye en el error acumulado:

$$\text{Secuencial : } ((s_1 + s_2) + s_3) + \dots \rightarrow \epsilon_{\text{seq}} = O(n \delta)$$

$$\text{Árbol paralelo (GPU) : } ((s_1 + s_2) + (s_3 + s_4)) + \dots \rightarrow \epsilon_{\text{tree}} \approx O(\log n \delta)$$

donde  $\delta$  es la máquina epsilon de punto flotante. Aunque ambas sumas son correctas en valor, el bit-exact resultado difiere ligeramente.

### Estrategia de Testing: 3-Capas

#### 1. CPU Baseline (Referencia):

```

1 os.environ['JAX_PLATFORMS'] = 'cpu'
2 result_cpu = predictor.predict_step(x_t, key)
3

```

Ejecutar con determinismo forzado en CPU. Este resultado se considera *ground truth*.

#### 2. GPU Determinista:

```

1 os.environ['JAX_PLATFORMS'] = 'gpu'
2 os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
3 os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':16:8'
4 # Con las mismas semillas y configuración XLA
5 result_gpu = predictor.predict_step(x_t, key)
6
7 assert jnp.allclose(result_cpu, result_gpu, atol=1e-7, rtol=1e-6)
8

```

Ejecutar en GPU con callbacks bloqueantes. La tolerancia se relaja ligeramente ( $10^{-7}$ ) debido a diferencias en el orden de operaciones.

#### 3. FPGA Simulada (Si aplicable):

```

1 # Simulación: forzar operaciones en float32 (FPGA typical) vs float64 (CPU)
2 jax.config.update('jax_enable_x64', False)
3 result_fpga = predictor.predict_step(x_t, key)
4 jax.config.update('jax_enable_x64', True)

```

```

5
6 # Tolerancia relavada para float32
7 assert jnp.allclose(result_cpu, result_fpga, atol=1e-4, rtol=1e-3)
8

```

### Procedimiento de Validación en CI/CD:

```

1 #!/bin/bash
2 # File: tests/determinism_test.sh
3
4 set -e
5
6 echo "[CPU] Running determinism test..."
7 export JAX_DETERMINISTIC_REDUCTIONS=1
8 export JAX_PLATFORMS=cpu
9 python tests/test_cpu_baseline.py
10
11 echo "[GPU] Running determinism test on GPU..."
12 export JAX_PLATFORMS=gpu
13 export CUDA_LAUNCH_BLOCKING=1
14 export CUBLAS_WORKSPACE_CONFIG=:16:8
15 python tests/test_gpu_consistency.py
16
17 echo "[FPGA Sim] Running determinism test with float32..."
18 python tests/test_fpga_sim.py
19
20 echo " All determinism tests passed"

```

### Notas de Configuración por Backend:

Backend	Variable de Entorno	Efecto
CPU	XLA_FLAGS=--xla_cpu_use_cross_replica_callbacks=false	Deshabilita callbacks paralelos
GPU (CUDA)	CUDA_LAUNCH_BLOCKING=1	Ejecuta kernels secuencialmente
GPU (CUDA)	CUBLAS_WORKSPACE_CONFIG=:16:8	Fijo workspace cuBLAS
Todos	JAX_DETERMINISTIC_REDUCTIONS=1	Fuerza orden de reducciones
Todos	JAX_DEFAULT_PRNG_IMPL=threefry2x32	PRNG portátil entre backends

### Garantías Post-Configuración:

- **Bit-Exactitud en CPU:** Garantizada si se ejecutan las mismas operaciones en el mismo orden.
- **Tolerancia GPU:**  $10^{-7}$  (atol) /  $10^{-6}$  (rtol) debido a paralelismo.
- **Tolerancia FPGA:**  $10^{-4}$  (atol) /  $10^{-3}$  (rtol) si usa float32.
- **PRNG:** Mismo seed produce igual secuencia en todos backends.

### Conclusión:

El determinismo en tests de portabilidad es crítico para *Pruebas.tex* (CPU vs GPU vs FPGA). Las variables de entorno XLA y la configuración de PRNG son **obligatorias** para CI/CD que valide consistencia entre plataformas.

## 15 Load Shedding Dinámico (Poda Topológica en Cisne Negro)

**Problema Crítico:** Durante eventos de cisne negro (ej. crash de mercado masivo en marzo 2024), la volatilidad  $\sigma_t$  puede explotar en múltiplos de  $5 - 20\times$  respecto al valor nominal. Esto desencadena dos patologías simultáneas:

1. **Saturación de Datos de Entrada:** La frecuencia de inyección de ticks sube de 1-10 Hz a 50-200 Hz (sistemas H.F.)
2. **Complejidad Factorial de Rama D:** La profundidad  $M = 5$  de log-signatures requiere  $\mathcal{O}(d^M)$  operaciones tensoriales, donde  $d$  es la dimensión del path

## Consecuencia sin Mitigación:

$$\text{Latencia de Inferencia} = T_{\text{kernel\_A+B+C}} + \underbrace{T_{\text{signature}}(M=5)}_{\text{Dominante: 80-120ms}} > \text{Tick-Rate de Entrada} = 5 - 20\text{ms}$$

Resultado: **Backlog infinito**, el sistema nunca se recupera, y las predicciones son obsoletas (stale) cuando finalmente se producen.

### Solución: Load Shedding Adaptativo en Orquestador

Incorporar lógica de **Deslastre de Carga Topológica** que monitoree la latencia de inferencia en tiempo real. Si la latencia media móvil (EWMA) supera el tick-rate de entrada, el sistema degrada dinámicamente:

$$M_{\text{efectivo}} = \begin{cases} 5 & \text{si EWMA(latencia)} < 0.7 \times \text{tick-rate} \\ 3 & \text{si } 0.7 \times \text{tick-rate} \leq \text{EWMA(latencia)} < \text{tick-rate} \\ 2 & \text{si EWMA(latencia)} \geq \text{tick-rate} \end{cases}$$

donde el umbral 0.7 introduce margen de seguridad (histéresis) para evitar oscilaciones.

### Implementación: Dual JIT Graph Switching

Precompile múltiples grafos JIT con diferentes profundidades  $M \in \{2, 3, 5\}$ :

```
1 import jax
2 import jax.numpy as jnp
3 from collections import deque
4 from typing import Dict, Callable
5
6 class AdaptiveSignatureOrchestrator:
7     """
8     Orquestrador con Load Shedding para Rama D.
9     Mantiene múltiples grafos JIT precompilados y selecciona dinámicamente.
10    """
11    def __init__(self, config: PredictorConfig):
12        self.config = config
13        self.tick_rate_ns = config.besov_nyquist_interval_ns # Mínimo tick esperado
14
15        # Precompilar grafos JIT para cada profundidad
16        self.signature_graphs: Dict[int, Callable] = {
17            2: jax.jit(lambda path: compute_signature(path, depth=2)),
18            3: jax.jit(lambda path: compute_signature(path, depth=3)),
19            5: jax.jit(lambda path: compute_signature(path, depth=5)),
20        }
21
22        # Monitoreo de latencia (EWMA)
23        self.latency_history = deque(maxlen=20) # Últimos 20 ticks
24        self.ewma_alpha = 0.3 # Factor de decaimiento
25        self.current_depth = 5 # Empezar en máxima resolución
26        self.ewma_latency_ns = 0.0
27
28        # Histéresis (evita flapping)
29        self.degradation_threshold = 0.7 # Degradar si latencia > 70% del tick-rate
30        self.recovery_threshold = 0.5 # Recuperar solo si latencia < 50% del tick-rate
31
32    def update_latency(self, inference_time_ns: int):
33        """Actualiza EWMA de latencia y decide si degradar/recuperar."""
34        self.latency_history.append(inference_time_ns)
35
36        # EWMA: L_t = alpha * L_actual + (1-alpha) * L_{t-1}
37        if self.ewma_latency_ns == 0.0:
38            self.ewma_latency_ns = float(inference_time_ns)
39        else:
40            self.ewma_latency_ns = (
41                self.ewma_alpha * inference_time_ns +
42                (1 - self.ewma_alpha) * self.ewma_latency_ns
43            )
44
45        # Decisión de degradación/recuperación
46        latency_ratio = self.ewma_latency_ns / self.tick_rate_ns
47
48        if latency_ratio >= 1.0: # Crítico: Ya sobrepasamos el tick-rate
```

```

49         self.current_depth = 2
50         elif latency_ratio >= self.degradation_threshold: # Advertencia
51             self.current_depth = min(self.current_depth, 3) # Degradar pero no al mínimo
52         elif latency_ratio < self.recovery_threshold and self.current_depth < 5:
53             # Recuperación gradual (histéresis)
54             self.current_depth = min(5, self.current_depth + 1)
55
56     def compute_adaptive_signature(self, path: jnp.ndarray) -> jnp.ndarray:
57         """
58         Calcula log-signature con profundidad adaptativa.
59         """
60         # Seleccionar grafo JIT según profundidad actual
61         signature_fn = self.signature_graphs[self.current_depth]
62
63         import time
64         t_start = time.perf_counter_ns()
65         result = signature_fn(path)
66         t_end = time.perf_counter_ns()
67
68         # Actualizar estadísticas
69         self.update_latency(t_end - t_start)
70
71         return result
72
73     def get_telemetry(self) -> Dict:
74         """Devuelve métricas de monitoreo."""
75         return {
76             "current_depth": self.current_depth,
77             "ewma_latency_ms": self.ewma_latency_ns / 1e6,
78             "tick_rate_ms": self.tick_rate_ns / 1e6,
79             "latency_ratio": self.ewma_latency_ns / self.tick_rate_ns,
80             "status": (
81                 "DEGRADED" if self.current_depth < 5 else "NOMINAL"
82             )
83         }

```

#### Análisis de Performance (Profundidad vs Latencia):

Profundidad M	Latencia Típica	Throughput (ticks/s)	Resolución Topológica
$M = 5$	80-120ms	8-12 Hz	Máxima (captura todas interacciones)
$M = 3$	20-35ms	28-50 Hz	Media (interacciones principales)
$M = 2$	5-12ms	83-200 Hz	Mínima (solo correlaciones simples)

#### Trade-off:

- **M=5:** Captura interacciones de orden 5 (ej.  $dx_1 \cdot dx_2 \cdot dx_3 \cdot dx_4 \cdot dx_5$ ) → Información topológica máxima pero  $\mathcal{O}(d^5)$  costo
- **M=3:** Captura hasta orden 3 (ej.  $dx_1 \cdot dx_2 \cdot dx_3$ ) → 3-4x más rápido
- **M=2:** Solo correlaciones pairwise → 10-15x más rápido, pero pierde estructura no-lineal profunda

#### Integración en UniversalPredictor:

```

1 class UniversalPredictor:
2     def __init__(self, config: PredictorConfig):
3         self.config = config
4         self.orchestrator = AdaptiveSignatureOrchestrator(config)
5         # ...resto de inicialización...
6
7     def predict_next(self, observation: MarketObservation) -> PredictionResult:
8         """
9         Predicción con Load Shedding automático.
10        """
11        # 1. Validar entrada
12        if not observation.validate_domain():
13            return self._fallback_prediction(observation)
14
15        # 2. Construir path incremental
16        path = self._build_path(observation)
17
18        # 3. Compute signature con profundidad adaptativa

```

```

19     signature = self.orchestrator.compute_adaptive_signature(path)
20
21     # 4. Resto de kernels (A, B, C)
22     # ...
23
24     # 5. Retornar con telemetría
25     telemetry = self.orchestrator.get_telemetry()
26     return PredictionResult(
27         predicted_next=predicted_value,
28         confidence=confidence_score,
29         metadata={
30             "signature_depth": telemetry["current_depth"],
31             "latency_ms": telemetry["ewma_latency_ms"],
32             "system_status": telemetry["status"]
33         }
34     )

```

### Validación en Cisne Negro (Simulación):

```

1 def test_black_swan_load_shedding():
2     """
3     Simula evento de cisne negro con tick-rate explosivo.
4     """
5     config = PredictorConfig(
6         log_sig_depth=5, # Profundidad nominal
7         besov_nyquist_interval_ns=100_000_000 # 100ms nominal
8     )
9     predictor = UniversalPredictor(config)
10
11     # Simular 1000 ticks con aceleración progresiva
12     tick_intervals = []
13     for i in range(1000):
14         if i < 100: # Primeros 100 ticks: normal (100ms)
15             tick_intervals.append(100_000_000)
16         elif i < 500: # Sigüientes 400 ticks: aceleración (20ms)
17             tick_intervals.append(20_000_000) # 50 Hz
18         else: # Últimos 500 ticks: crisis (5ms)
19             tick_intervals.append(5_000_000) # 200 Hz
20
21     depths_observed = []
22     latencies_observed = []
23
24     for interval_ns in tick_intervals:
25         observation = MarketObservation(
26             price=jnp.array([random.normal()]),
27             target=jnp.array([0.0]),
28             timestamp_ns=time.time_ns()
29         )
30
31         result = predictor.predict_next(observation)
32         depths_observed.append(result.metadata["signature_depth"])
33         latencies_observed.append(result.metadata["latency_ms"])
34
35         time.sleep(interval_ns / 1e9) # Simular tick-rate
36
37     # Assertions
38     assert depths_observed[50] == 5, "Debe estar en M=5 durante régimen normal"
39     assert depths_observed[300] == 3, "Debe degradar a M=3 durante aceleración"
40     assert depths_observed[700] == 2, "Debe degradar a M=2 durante crisis"
41
42     # Verificar que latencia nunca causa backlog infinito
43     max_backlog_ratio = max(
44         latencies_observed[i] / (tick_intervals[i] / 1e6)
45         for i in range(len(tick_intervals))
46     )
47     assert max_backlog_ratio < 1.2, f"Backlog ratio demasiado alto: {max_backlog_ratio}"
48
49     print(" [] Load Shedding test PASSED")
50     print(f"     Profundidad nominal: M=5")
51     print(f"     Profundidad mínima observada: M={min(depths_observed)}")
52     print(f"     Latencia máxima: {max(latencies_observed):.2f}ms")
53     print(f"     Max backlog ratio: {max_backlog_ratio:.2f}x")

```

### Beneficios en Producción:

- **Prevención de Backlog Infinito:** Sistema siempre se adapta al tick-rate, nunca colapsa
- **Graceful Degradation:** Reduce resolución topológica pero mantiene predicciones (vs fallo total)
- **Recuperación Automática:** Cuando volatilidad baja, sistema escala de vuelta a  $M = 5$
- **Overhead Mínimo:** Decisión de switching toma  $< 0.1\text{ms}$  (simple comparación EWMA)
- **Visibilidad:** Telemetría expone profundidad actual en cada predicción

#### Consideraciones Operacionales:

1. **Alertas de Degradación:** Si sistema permanece en  $M < 5$  por más de 60 segundos, emitir alerta de alta volatilidad
2. **Persistencia de Estado:** El `AdaptiveSignatureOrchestrator` debe incluirse en snapshots para preservar EWMA tras restart
3. **Configuración de Umbrales:** Los thresholds 0.7 y 0.5 deben ser ajustables vía `PredictorConfig`
4. **Métricas de Monitoreo:** Exponer `current_depth`, `ewma_latency_ms`, `latency_ratio` a Prometheus/-Grafana

#### Alternativa Avanzada: Poda Selectiva de Kernels

En lugar de solo degradar Rama D, se puede escalar a deshabilitar temporalmente kernels completos:

```

1 # Modo ultra-degradado (solo Kernel A + Rama D superficial)
2 if latency_ratio > 2.0:
3     result = kernel_a_only_prediction(observation) # Rama A sin Signatures
4 elif latency_ratio > 1.0:
5     result = kernels_a_b_only(observation) # Rama A + B (sin Neural ODE Rama C)
6 else:
7     result = full_prediction(observation) # Todas las Ramas

```

Esto permite **mantener throughput** incluso en crisis extremas (ej. flash crash de mayo 2010).

#### Conclusión:

El Load Shedding Dinámico es **esencial** para sistemas H.F. que operan en mercados volátiles. Sin esta lógica, eventos de cisne negro producirían backlog infinito y obsolescencia de predicciones. La degradación topológica ( $M = 5 \rightarrow 3 \rightarrow 2$ ) sacrifica resolución no-lineal profunda pero garantiza que el sistema **nunca se sature**, priorizando latencia sobre precisión extrema cuando es necesario.

## 16 Telemetría de Jitter (Gatillo Preciso para Load Shedding)

#### Problema de Precisión en Detección de Saturación:

El Load Shedding descrito en la sección anterior utiliza **EWMA de latencia media** como gatillo. Sin embargo, existe un problema sutil:

$$\text{EWMA}(\text{latencia}) = \alpha \cdot L_t + (1 - \alpha) \cdot \text{EWMA}_{t-1}$$

Esta métrica captura la *tendencia central* de la latencia, pero **ignora la variabilidad** (jitter). En un sistema que procesa frames de mercado a frecuencia variable, la *varianza* de la latencia es crítica:

#### Escenario Problemático:

- Tick #100: Latencia = 0.8ms (rápido, Rama D con  $M = 5$ )
- Tick #101: Latencia = 22ms (lento, GC de JAX o swap de memoria)
- Tick #102: Latencia = 0.9ms (rápido nuevamente)
- Tick #103: Latencia = 25ms (lento, conflicto de I/O)
- ...



**EWMA observado:**  $\sim 6\text{--}8\text{ms}$  (promedio) **Jitter observado:**  $\sigma_{\text{latencia}} \sim 10\text{--}12\text{ms}$  (desviación estándar)

El sistema parece *nominalmente aceptable* según EWMA, pero el **jitter catastrófico** indica que el sistema está *al borde del colapso*. En algunos ticks excede 20ms  $\rightarrow$  backlog se acumula  $\rightarrow$  predicciones obsoletas.

#### Criterio de Nyquist para Jitter:

En teoría de muestreo, el límite de Nyquist establece que para reconstruir una señal con frecuencia  $f$ , se requiere muestreo a  $\geq 2f$ . En nuestro contexto:

$$\text{Tick-Rate}_{\text{entrada}} = \frac{1}{\text{besov\_nyquist\_interval\_ns}}$$

Si el **jitter de latencia** (varianza) excede el 80% del tick-rate, el sistema está en riesgo de perder sincronización:

$$\text{Jitter}_{\text{threshold}} = 0.8 \times \text{besov\_nyquist\_interval\_ns}$$

**Razón:** Si la desviación estándar de latencia es  $\sim 80\text{ms}$  y el tick-rate es 100ms, entonces:

- $\mu_{\text{latencia}} - \sigma_{\text{latencia}} = 50 - 80 = -30\text{ms}$  (algunos ticks son instantáneos)
- $\mu_{\text{latencia}} + \sigma_{\text{latencia}} = 50 + 80 = 130\text{ms}$  (algunos ticks exceden el tiempo disponible)

Esto garantiza que al menos algunas predicciones serán *obsoletas*, violando SLA de latencia.

#### Solución: Telemetría de Alta Resolución con perf\_counter\_ns

Python 3.7+ proporciona `time.perf_counter_ns()`, un reloj monotónico de alta resolución (nanosegundos) que no se ve afectado por ajustes de NTP o cambios de zona horaria:

```

1 import time
2 import jax.numpy as jnp
3 from collections import deque
4 from typing import Deque, Dict
5
6 class JitterMonitor:
7     """
8     Monitor de varianza de latencia (jitter) para detección precisa de saturación.
9     Complementa el EWMA de latencia con métrica de dispersión.
10    """
11    def __init__(self,
12                 window_size: int = 100,
13                 nyquist_interval_ns: int = 100_000_000, # 100ms
14                 jitter_threshold_ratio: float = 0.8):
15        """
16        Args:
17            window_size: Tamaño de ventana deslizante para calcular jitter
18            nyquist_interval_ns: Intervalo mínimo entre ticks (Nyquist limit)
19            jitter_threshold_ratio: Umbral de jitter como fracción del Nyquist (ej. 0.8 = 80%)
20        """
21        self.window_size = window_size
22        self.nyquist_interval_ns = nyquist_interval_ns
23        self.jitter_threshold_ns = jitter_threshold_ratio * nyquist_interval_ns
24
25        # Historial de latencias (ventana deslizante)
26        self.latency_history: Deque[int] = deque(maxlen=window_size)
27
28        # Estadísticas acumuladas
29        self.total_predictions = 0
30        self.jitter_violations = 0
31        self.max_latency_observed_ns = 0
32
33    def record_latency(self, latency_ns: int) -> Dict:
34        """
35        Registra latencia de predicción y calcula jitter.
36
37        Returns:
38            Dict con métricas: jitter_ns, mean_ns, std_ns, violation, etc.
39        """
40        self.latency_history.append(latency_ns)
41        self.total_predictions += 1
42        self.max_latency_observed_ns = max(self.max_latency_observed_ns, latency_ns)

```

```

43
44     # Calcular estadísticas solo si hay suficiente historia
45     if len(self.latency_history) < 10:
46         return {
47             "jitter_ns": 0,
48             "mean_ns": latency_ns,
49             "std_ns": 0,
50             "violation": False,
51             "degradation_recommended": False
52         }
53
54     # Convertir a array numpy para cálculo eficiente
55     latencias = jnp.array(list(self.latency_history), dtype=jnp.float64)
56
57     mean_latency = jnp.mean(latencias)
58     std_latency = jnp.std(latencias) # Desviación estándar = jitter
59
60     # Criterio de violación: jitter > 80% del tick-rate
61     violation = std_latency > self.jitter_threshold_ns
62
63     if violation:
64         self.jitter_violations += 1
65
66     # Recomendación de degradación (combina jitter + media)
67     degradation_recommended = (
68         violation or
69         (mean_latency > 0.7 * self.nyquist_interval_ns and std_latency > 0.5 * self.
70         nyquist_interval_ns)
71     )
72
73     return {
74         "jitter_ns": float(std_latency),
75         "mean_ns": float(mean_latency),
76         "std_ns": float(std_latency),
77         "violation": bool(violation),
78         "degradation_recommended": bool(degradation_recommended),
79         "nyquist_limit_ns": self.nyquist_interval_ns,
80         "jitter_threshold_ns": self.jitter_threshold_ns,
81         "jitter_ratio": float(std_latency / self.nyquist_interval_ns) if self.
82         nyquist_interval_ns > 0 else 0.0
83     }
84
85     def get_summary_stats(self) -> Dict:
86         """Devuelve resumen estadístico completo."""
87         if len(self.latency_history) == 0:
88             return {"status": "NO_DATA"}
89
90         latencias = jnp.array(list(self.latency_history), dtype=jnp.float64)
91
92         return {
93             "total_predictions": self.total_predictions,
94             "jitter_violations": self.jitter_violations,
95             "violation_rate": self.jitter_violations / self.total_predictions if self.
96             total_predictions > 0 else 0.0,
97             "mean_latency_ms": float(jnp.mean(latencias) / 1e6),
98             "std_latency_ms": float(jnp.std(latencias) / 1e6),
99             "min_latency_ms": float(jnp.min(latencias) / 1e6),
100             "max_latency_ms": float(jnp.max(latencias) / 1e6),
101             "p50_latency_ms": float(jnp.percentile(latencias, 50) / 1e6),
102             "p95_latency_ms": float(jnp.percentile(latencias, 95) / 1e6),
103             "p99_latency_ms": float(jnp.percentile(latencias, 99) / 1e6),
104             "max_latency_ever_ms": float(self.max_latency_observed_ns / 1e6)
105         }

```

### Integración con Load Shedding:

El JitterMonitor complementa al AdaptiveSignatureOrchestrator, proporcionando señal de degradación más precisa:

```

1 class AdaptiveSignatureOrchestrator_WithJitter:
2     """
3     Versión mejorada con telemetría de jitter.
4     """
5     def __init__(self, config: PredictorConfig):
6         self.config = config

```

```

7     self.tick_rate_ns = config.besov_nyquist_interval_ns
8
9     # Orquestador original (EWMA de latencia)
10    self.signature_graphs = {
11        2: jax.jit(lambda path: compute_signature(path, depth=2)),
12        3: jax.jit(lambda path: compute_signature(path, depth=3)),
13        5: jax.jit(lambda path: compute_signature(path, depth=5)),
14    }
15    self.current_depth = 5
16    self.ewma_latency_ns = 0.0
17    self.ewma_alpha = 0.3
18
19    # NUEVO: Monitor de jitter
20    self.jitter_monitor = JitterMonitor(
21        window_size=100,
22        nyquist_interval_ns=self.tick_rate_ns,
23        jitter_threshold_ratio=0.8
24    )
25
26    # Histéresis mejorada
27    self.degradation_threshold = 0.7
28    self.recovery_threshold = 0.5
29
30    def compute_adaptive_signature(self, path: jnp.ndarray) -> jnp.ndarray:
31        """
32        Calcula signature con profundidad adaptativa basada en EWMA + JITTER.
33        """
34        signature_fn = self.signature_graphs[self.current_depth]
35
36        # Medir latencia con reloj de alta resolución
37        t_start = time.perf_counter_ns()
38        result = signature_fn(path)
39        jax.block_until_ready(result) # Forzar sincronización GPU
40        t_end = time.perf_counter_ns()
41
42        latency_ns = t_end - t_start
43
44        # Actualizar EWMA (como antes)
45        if self.ewma_latency_ns == 0.0:
46            self.ewma_latency_ns = float(latency_ns)
47        else:
48            self.ewma_latency_ns = (
49                self.ewma_alpha * latency_ns +
50                (1 - self.ewma_alpha) * self.ewma_latency_ns
51            )
52
53        # NUEVO: Actualizar monitor de jitter
54        jitter_metrics = self.jitter_monitor.record_latency(latency_ns)
55
56        # DECISIÓN DE DEGRADACIÓN (criterio combinado: EWMA + Jitter)
57        latency_ratio = self.ewma_latency_ns / self.tick_rate_ns
58        jitter_violation = jitter_metrics["violation"]
59        jitter_ratio = jitter_metrics["jitter_ratio"]
60
61        # Degradación agresiva si jitter es alto
62        if jitter_violation and jitter_ratio > 1.0:
63            # Jitter crítico: degradar directamente a M=2
64            self.current_depth = 2
65        elif latency_ratio >= 1.0 or jitter_metrics["degradation_recommended"]:
66            # Latencia crítica o jitter recomendado: degradar a M=2
67            self.current_depth = 2
68        elif latency_ratio >= self.degradation_threshold or jitter_ratio > 0.6:
69            # Advertencia: degradar a M=3
70            self.current_depth = min(self.current_depth, 3)
71        elif latency_ratio < self.recovery_threshold and jitter_ratio < 0.3:
72            # Recuperación: ambos indicadores saludables
73            self.current_depth = min(5, self.current_depth + 1)
74
75        return result
76
77    def get_telemetry(self) -> Dict:
78        """Devuelve métricas combinadas: EWMA + Jitter."""
79        jitter_summary = self.jitter_monitor.get_summary_stats()

```

```

80
81     return {
82         "current_depth": self.current_depth,
83         "ewma_latency_ms": self.ewma_latency_ns / 1e6,
84         "tick_rate_ms": self.tick_rate_ns / 1e6,
85         "latency_ratio": self.ewma_latency_ns / self.tick_rate_ns,
86         "jitter_ms": jitter_summary.get("std_latency_ms", 0.0),
87         "jitter_ratio": (jitter_summary.get("std_latency_ms", 0.0) * 1e6) / self.
tick_rate_ns,
88         "jitter_violations": jitter_summary.get("jitter_violations", 0),
89         "violation_rate": jitter_summary.get("violation_rate", 0.0),
90         "p99_latency_ms": jitter_summary.get("p99_latency_ms", 0.0),
91         "status": (
92             "DEGRADED_CRITICAL" if self.current_depth == 2 else
93             "DEGRADED" if self.current_depth < 5 else
94             "NOMINAL"
95         )
96     }

```

### Análisis de Métricas Combinadas:

Escenario	EWMA (ms)	Jitter (ms)	Decisión	M
Nominal estable	0.8	0.1	Mantener	5
Alta media, bajo jitter	12.0	2.0	Degradar (EWMA)	3
Baja media, alto jitter	5.0	25.0	Degradar (Jitter)	2
Crisis combinada	18.0	30.0	Degradar crítico	2
Recuperación gradual	6.0	3.0	Mantener degradado	3
Recuperación completa	0.9	0.2	Escalar a nominal	5

### Ventajas del Criterio Dual (EWMA + Jitter):

1. **Detección Temprana:** Jitter alto detecta problemas antes de que EWMA cruce umbral
2. **Robustez a Outliers:** EWMA suaviza spikes, jitter los captura
3. **Prevención de Flapping:** Criterio de recuperación requiere AMBOS indicadores saludables
4. **Granularidad:** Jitter permite 3 niveles de degradación (5→3→2) en vez de binario
5. **Visibilidad Operacional:** P99 latency expuesto para alertas de producción

### Implementación de Medición Precisa:

```

1 class UniversalPredictor_WithJitterMonitoring:
2     def __init__(self, config: PredictorConfig):
3         self.config = config
4         self.orchestrator = AdaptiveSignatureOrchestrator_WithJitter(config)
5         # ...resto de inicialización...
6
7     def predict_next(self, observation: MarketObservation) -> PredictionResult:
8         """
9         Predicción con telemetría de jitter de alta resolución.
10        """
11        # Tiempo de predicción TOTAL (incluye todos los kernels)
12        t_prediction_start = time.perf_counter_ns()
13
14        # 1. Validar entrada
15        if not observation.validate_domain():
16            return self._fallback_prediction(observation)
17
18        # 2. Construir path incremental
19        path = self._build_path(observation)
20
21        # 3. Compute signature con profundidad adaptativa (ya mide internamente)
22        signature = self.orchestrator.compute_adaptive_signature(path)
23
24        # 4. Resto de kernels (A, B, C)
25        # ...
26
27        # 5. Orquestador (fusión + JK0)

```

```

28     predicted_value = self.orchestrator.predict_step(observation.price)
29
30     # MEDIR LATENCIA TOTAL
31     t_prediction_end = time.perf_counter_ns()
32     total_latency_ns = t_prediction_end - t_prediction_start
33
34     # Obtener telemetría
35     telemetry = self.orchestrator.get_telemetry()
36
37     # Logging condicional (solo si jitter es alto)
38     if telemetry["jitter_ratio"] > 0.6:
39         self.logger.warning(
40             f"[JITTER ALERT] Ratio={telemetry['jitter_ratio']:.2f}, "
41             f"P99={telemetry['p99_latency_ms']:.2f}ms, "
42             f"Depth={telemetry['current_depth']}"
43         )
44
45     return PredictionResult(
46         predicted_next=predicted_value,
47         confidence=self._compute_confidence(signature),
48         metadata={
49             "signature_depth": telemetry["current_depth"],
50             "ewma_latency_ms": telemetry["ewma_latency_ms"],
51             "jitter_ms": telemetry["jitter_ms"],
52             "jitter_ratio": telemetry["jitter_ratio"],
53             "p99_latency_ms": telemetry["p99_latency_ms"],
54             "total_latency_ns": total_latency_ns,
55             "system_status": telemetry["status"]
56         }
57     )

```

### Test de Jitter con Carga Variable:

```

1 def test_jitter_detection_under_variable_load():
2     """
3     Simula carga con latencia variable (jitter) para validar detección.
4     """
5     import random
6
7     config = PredictorConfig(
8         log_sig_depth=5,
9         besov_nyquist_interval_ns=100_000_000 # 100ms
10    )
11    predictor = UniversalPredictor_WithJitterMonitoring(config)
12
13    # Simular 200 predicciones con patrones de latencia diferentes
14    latencies_observed = []
15    depths_observed = []
16
17    for i in range(200):
18        # FASE 1 (i < 50): Latencia estable (0.8-1.2ms)
19        if i < 50:
20            simulated_processing_time_ns = random.uniform(800_000, 1_200_000)
21
22        # FASE 2 (50 <= i < 100): Jitter alto (0.5-30ms)
23        elif i < 100:
24            simulated_processing_time_ns = random.uniform(500_000, 30_000_000)
25
26        # FASE 3 (100 <= i < 150): Latencia alta constante (15-20ms)
27        elif i < 150:
28            simulated_processing_time_ns = random.uniform(15_000_000, 20_000_000)
29
30        # FASE 4 (150 <= i < 200): Recuperación gradual (1-3ms)
31        else:
32            simulated_processing_time_ns = random.uniform(1_000_000, 3_000_000)
33
34        # Simular predicción
35        observation = MarketObservation(
36            price=jnp.array([random.normal()]),
37            target=jnp.array([0.0]),
38            timestamp_ns=time.time_ns()
39        )
40
41        # Inyectar latencia artificial

```

```

42         time.sleep(simulated_processing_time_ns / 1e9)
43
44         result = predictor.predict_next(observation)
45
46         latencies_observed.append(result.metadata["total_latency_ns"] / 1e6)
47         depths_observed.append(result.metadata["signature_depth"])
48
49     # Assertions
50     # FASE 1: Sistema debe mantener M=5 (jitter bajo)
51     assert all(d == 5 for d in depths_observed[10:50]), "Debería mantener M=5 en fase estable"
52
53     # FASE 2: Sistema debe degradar por jitter alto
54     assert any(d < 5 for d in depths_observed[60:100]), "Debería degradar por jitter alto"
55
56     # FASE 3: Sistema debe degradar por latencia alta
57     assert all(d <= 3 for d in depths_observed[120:150]), "Debería degradar a M<=3 por latencia alta"
58
59     # FASE 4: Sistema debe recuperar gradualmente
60     final_depth = depths_observed[-10:]
61     assert any(d >= 4 for d in final_depth), "Debería recuperar hacia M=5"
62
63     # Estadísticas finales
64     telemetry = predictor.orchestrator.get_telemetry()
65
66     print(" [] Jitter detection test PASSED")
67     print(f" Jitter violations: {telemetry['jitter_violations']}")
68     print(f" Violation rate: {telemetry['violation_rate']:.2%}")
69     print(f" P99 latency: {telemetry['p99_latency_ms']:.2f}ms")
70     print(f" Final depth: {telemetry['current_depth']}")
71     print(f" Final jitter ratio: {telemetry['jitter_ratio']:.2f}")

```

### Dashboard de Monitoreo (Prometheus + Grafana):

```

1 from prometheus_client import Histogram, Gauge, Counter
2
3 class UniversalPredictor_WithMetrics:
4     def __init__(self, config: PredictorConfig):
5         # ...inicialización...
6
7         # Métricas de latencia
8         self.latency_histogram = Histogram(
9             'predictor_latency_seconds',
10            'Latency distribution of predictions',
11            buckets=[0.001, 0.005, 0.010, 0.050, 0.100, 0.500, 1.0]
12        )
13
14        # Métricas de jitter
15        self.jitter_gauge = Gauge(
16            'predictor_jitter_milliseconds',
17            'Standard deviation of latency (jitter)'
18        )
19        self.jitter_ratio_gauge = Gauge(
20            'predictor_jitter_ratio',
21            'Jitter as ratio of Nyquist limit'
22        )
23        self.jitter_violations_counter = Counter(
24            'predictor_jitter_violations_total',
25            'Total number of jitter threshold violations'
26        )
27
28        # Métricas de profundidad
29        self.signature_depth_gauge = Gauge(
30            'predictor_signature_depth',
31            'Current signature computation depth (M)'
32        )
33
34    def predict_next(self, observation: MarketObservation) -> PredictionResult:
35        with self.latency_histogram.time():
36            result = self._predict_internal(observation)
37
38        # Actualizar métricas de jitter
39        telemetry = self.orchestrator.get_telemetry()
40        self.jitter_gauge.set(telemetry["jitter_ms"])

```

```

41     self.jitter_ratio_gauge.set(telemetry["jitter_ratio"])
42     self.signature_depth_gauge.set(telemetry["current_depth"])
43
44     if telemetry["jitter_ratio"] > 0.8:
45         self.jitter_violations_counter.inc()
46
47     return result

```

### Alertas Operacionales (Recomendadas):

```

1 # prometheus_alerts.yml
2 groups:
3 - name: predictor_latency
4   interval: 30s
5   rules:
6   - alert: PredictorHighJitter
7     expr: predictor_jitter_ratio > 0.8
8     for: 1m
9     labels:
10      severity: warning
11     annotations:
12      summary: "High jitter detected in predictor"
13      description: "Jitter ratio {{ $value }} exceeds 80% of Nyquist limit"
14
15   - alert: PredictorJitterCritical
16     expr: predictor_jitter_ratio > 1.2
17     for: 30s
18     labels:
19      severity: critical
20     annotations:
21      summary: "Critical jitter in predictor"
22      description: "Jitter ratio {{ $value }} exceeds Nyquist limit, predictions may be stale"
23
24   - alert: PredictorDegraded
25     expr: predictor_signature_depth < 5
26     for: 5m
27     labels:
28      severity: info
29     annotations:
30      summary: "Predictor operating in degraded mode"
31      description: "Signature depth reduced to {{ $value }} due to load"

```

### Conclusión:

La **Telemetría de Jitter** es complemento esencial al Load Shedding basado en EWMA:

- **EWMA solo:** Captura tendencia central, pero *ignora* volatilidad
- **Jitter solo:** Detecta variabilidad, pero puede ser *ruidoso* en régimen estable
- **EWMA + Jitter:** Combinación robusta que detecta *tanto* sobrecarga sostenida como spikes intermitentes

El uso de `time.perf_counter_ns()` garantiza precisión de nanosegundos sin overhead significativo (<50ns por llamada). El criterio de umbral al 80% del límite de Nyquist proporciona margen de seguridad antes de que el backlog se vuelva irrecuperable.

En producción, esta telemetría debe exponerse a Prometheus/Grafana para:

1. Dashboards en tiempo real de salud del sistema
2. Alertas automáticas cuando jitter excede umbrales
3. Análisis post-mortem de incidentes de latencia
4. Ajuste dinámico de `jitter_threshold_ratio` según carga de mercado

Sin monitoreo de jitter, el sistema puede operar en *modo degradado silencioso*, donde algunas predicciones son rápidas y otras lentas, promediando a un valor *aparentemente* aceptable, pero violando SLA de latencia en percentiles altos (P95, P99).