

Especificación de API Python - Universal Predictor

Ingeniería de Software

February 18, 2026

Contents

1	Introducción	3
2	Estructuras de Datos (Tipado)	3
2.1	Configuración (Λ)	3
2.2	Entrada Operativa (y_t, y_{target}, τ)	3
2.3	Salida del Sistema	3
3	Arquitectura Multitenencia (Stateless/Functional Pattern)	4
3.1	Maximización de Throughput (Batching Vectorizado)	4
4	Clase Principal: UniversalPredictor (Stateful Wrapper)	5
4.1	Inicialización	5
4.2	Método de Ejecución (Paso $t \rightarrow t + 1$)	5
5	Prevención de Fragmentación de VRAM (JAX Memory Management)	6
6	Persistencia (Atomic Snapshotting)	9
7	I/O Asíncrono para Snapshots (Non-Blocking)	10
8	Ajuste Adaptativo del Umbral CUSUM	13
8.1	Fórmula de Ajuste	13
8.2	Interpretación de Curtosis	13
9	Periodo de Gracia (Ventana Refractaria) Post-Cambio de Régimen	14
9.1	Motivación	14
9.2	Solución: Grace Period (Refractario)	14
9.3	Algoritmo de Implementación	14
9.4	Parámetros Sugeridos	15
9.5	Diagnóstico y Telemetría	15
10	Flags de Operación y Recuperación	15
10.1	DegradedInferenceMode	15
10.2	EmergencyMode	16
10.3	RegimeChangeDetected	16
10.4	ModeCollapseWarning	16
10.5	Período de Gracia (Refractario) Post-Cambio de Régimen	16
10.5.1	Implementación	16
10.5.2	Dinámica Temporal	16
10.5.3	Parámetros Recomendados	17
10.5.4	Ventajas	17
11	Manejo de Errores y Excepciones	17
11.1	Excepciones Estándar	17
11.2	Alertas Específicas Avanzadas	18
11.3	Ejemplo de Logging en Producción	18

12 Detección de Mode Collapse en DGM	19
12.1 Criterio de Detección	19
12.2 Acción Correctiva	19
13 Determinismo de Punto Flotante (Bit-Exact Reproducibility)	19
14 Load Shedding Dinámico (Poda Topológica en Cisne Negro)	22

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 (Λ)

```
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 JK0
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
23    WTMM
24    inference_recovery_hysteresis: float = 0.8 # Factor histéresis para recuperación de modo degradado
```

2.2 Entrada Operativa (y_t, y_{target}, τ)

```
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_toCollapse: Float[Array, "1"] # h - G^+
9     free_energy: Float[Array, "1"]       # F (Energía JK0)
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     modeCollapse_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 injectado 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 parallelizar 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)
50         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 (residente en GPU)
13         self._jit_update = jax.jit(self._core_update_step)
13         self._last_timestamp_ns = 0 # Para cálculo de frecuencia
14
15     def fit_history(self, history: list[float]) -> bool:
16         """
17             Bootstrapping inicial (Protocolo de Cold Start).
18             Procesa el lote histórico para estabilizar los pesos JKO y llenar los búferes.
19             Requiere un mínimo de N_buf muestras.
20
21             Returns:
22                 bool: True si el sistema alcanzó convergencia estable (Sinkhorn + CUSUM).
23             Raises:
24                 ValueError: Si el historial es insuficiente (< wtmm_buffer_size).
25                 RuntimeError: Si el sistema diverge tras el calentamiento.
26         """
27         if len(history) < self.config.wtmm_buffer_size:
28             raise ValueError(f"Historial insuficiente. Requerido: {self.config.
28 wtmm_buffer_size}")
29
30         # Ejecución batch acelerada (jax.lax.scan) para calentar el estado
31         # Simula el paso del tiempo para llenar colas y estabilizar gradientes
32         self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))
33
34         # Validación de Convergencia
35         is_converged = final_metrics.sinkhorn_converged
36         is_stable = final_metrics.cusum_drift < self.config.cusum_h
37
38         if not (is_converged and is_stable):
39             logger.warning("Cold Start finalizado sin convergencia estable.")
40             return False
41
42         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.
19     besov_nyquist_interval_ns)
20
21     if is_sparse:
22         logger.warning(f"FrequencyWarning: Event interval {dt_arrival}ns > Nyquist limit.
23 WTMM spectrum might alias.")
24
25     self._last_timestamp_ns = obs.timestamp_ns
26
27     # 3. Actualización Core (JAX) - Zero-Copy State Management
28     # IMPORTANTE: El buffer de señal reside en GPU/TPU (self._state.signal_buffer).
29     # La actualización se realiza "in-place" funcionalmente usando jax.lax.
30     dynamic_update_slice
31     # o jnp.roll dentro del kernel compilado para evitar transferencias CPU <-> VRAM.
32     # Si hay staleness o sparsity excesiva, se congelan pesos para no degradar la geometri
33     a.
34     should_freeze = is_stale or is_sparse
35
36     new_state, result_data = self._jit_update(
37         self._state,
38         obs.price,
39         obs.target,
40         freeze_weights=should_freeze,
41         # No se pasa history_buffer explícitamente, ya vive en _state
42     )
43
44     self._state = new_state
45
46     # 4. Empaqueado de Resultados
47     return PredictionResult(
48         predicted_next=result_data.y_next,
49         holder_exponent=result_data.H_t,
50         sinkhorn_converged=result_data.converged,
51         is_stable=not (is_stale or is_sparse),
52         # ... mapeo resto de campos
53     )

```

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/lanzan 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: Usarallocador "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: {os.environ['XLA_PYTHON_CLIENT_ALLOCATOR'])}
```

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,nounits,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
10        # Serialización de tensores JAX a bytes
11        state_dict = self._serialize_jax_state(self._state)
12
13        # Segmentación Modular (K-Blocks)
14        # IMPORTANTE: Incluir versionado de schema para evitar errores al cargar
15        # snapshots generados con versiones antiguas (cambios en profundidad de firma, etc.)
16        payload = {
17            "schema_version": self.config.schema_version, # Versionado seguro
18            "timestamp": time.time_ns(),
19            "config": asdict(self.config),
20            "global": state_dict["global"], # rho, G+, ema
21            "telemetry": {
22                "kurtosis": float(self._state.kurtosis),
23                "dgm_entropy": float(self._state.dgm_entropy),
24                "adaptive_threshold": float(self._state.h_adaptive)
25            },
26            "flags": {
27                "degraded_inference": bool(self._state.degraded_mode),
28                "emergency": bool(self._state.emergency_mode),
29                "regime_change": bool(self._state.regime_changed),
30                "mode_collapse": bool(self._state.mode_collapse_warning)
31            },
32            "kernels": {
33                "A": state_dict["kernel_a"],
34                "B": state_dict["kernel_b"],
35                "C": state_dict["kernel_c"],
36                "D": state_dict["kernel_d"]
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
70             features."
71         )
72
73     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
25        checksum = hashlib.sha256(data_bytes).hexdigest()
26
27        # Escritura atómica (write + rename) para evitar estado intermedio
28        temp_filepath = filepath + ".tmp"
29
30        try:
31            with open(temp_filepath, "wb") as f:
32                f.write(data_bytes)
33                f.write(checksum.encode('utf-8'))
34
35            # Rename atómico (POSIX-compliant)
36            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,
110         Optional[concurrent.futures.Future]]:
111         """
112             Paso de predicción con snapshot asincrónico periódico.
113
114             El snapshot se dispara cada N pasos pero NO interfiere con latencia de inferencia.
115             """
116             # Predicción principal (hot path)
117             prediction = self._predict_step_core(x_t)
118
119             # Checkpoint asincrónico si es el momento
120             snapshot_future = None
121             if self.step_counter % self.checkpoint_interval == 0:
122                 checkpoint_path = f"{self.checkpoint_dir}/snapshot_step_{self.step_counter}.msgpack"
123                 snapshot_future = self.save_snapshot_nonblocking(checkpoint_path)
124                 # El hilo retorna SIN esperar a que el snapshot se escriba a disco
125
126             self.step_counter += 1
127             return prediction, snapshot_future
128
129     def monitor_snapshot_queue(self):
130         """
131             Thread de monitoreo (opcional) que reporta el estado de snapshots en vuelo.
132             Ejecutado en otro thread para no interferir.
133             """
134             while not self.shutdown_event.wait(timeout=5.0):
135                 with self.snapshot_lock:
136                     if self.pending_snapshot_future is not None:
137                         if self.pending_snapshot_future.done():
138                             try:
139                                 result = self.pending_snapshot_future.result()
140                                 if result['status'] == 'success':
141                                     print(f"[INFO] Snapshot guardado: {result['filepath']} "
142                                         f"({result['filesize_bytes']} bytes)")
143                                 else:
144                                     print(f"[WARNING] Snapshot falló: {result['error']}")
145                         except Exception as e:
146                             print(f"[ERROR] Future exception: {e}")
147
148     def graceful_shutdown(self, timeout_seconds=10):
149         """
150             Aguarda a que todos los snapshots pendientes se completen antes de cerrar.
151             """
152             print("[INFO] Aguardando snapshots pendientes...")
153
154             # Esperar a que se completen (con timeout)
155             concurrent.futures.wait(
156                 [self.pending_snapshot_future] if self.pending_snapshot_future else [],
157                 timeout=timeout_seconds
158             )
159
160             # Cerrar pool
161             self.io_executor.shutdown(wait=True)
162             print("[INFO] ThreadPoolExecutor cerrado gracefully")

```

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)
Latencia observed por predictor	0.1–0.5	Principal

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

Garantías Operacionales:

- **Fire-and-Forget:** Ignorar el Future returnedo 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 ≤ 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 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 régimenes con colas pesadas.

8.1 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)
- σ_t : Volatilidad EMA del error de predicción
- κ_t : Curtosis empírica móvil (ventana de 252 pasos)
- 3: Curtosis de referencia Gaussiana

8.2 Interpretación de Curtosis

Rango κ_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 régimenes de alta curtosis mientras mantiene sensibilidad a cambios estructurales genuinos.

9 Periodo de Gracia (Ventana Refractaria) Post-Cambio de Régimen

9.1 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.

9.2 Solución: Grace Period (Refractario)

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

`CUSUM_silenciado = (pasos_desde_último_cambio < 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

9.3 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

```

9.4 Parámetros Sugeridos

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 líquidos o con gaps	Minimizar oscilaciones patológicas

Table 2: Recomendaciones para grace_period_steps según régimen

9.5 Diagnóstico y Telemetría

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

- G^+ observable (si hubiera alarma)
- σ_t y κ_t instantáneos
- Convergencia del JKO (distancia Wasserstein a cada paso)

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

10 Flags de Operación y Recuperación

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

10.1 DegradedInferenceMode

Condición de activación:

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

Implicaciones operacionales:

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

Recuperación con histéresis:

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

donde $h_{\text{hyt}} = \text{inference_recovery_hysteresis}$ (por defecto 0.8) parametrizable en PredictorConfig.

Se emite NormalOperationRestoredEvent al recuperar.

10.2 EmergencyMode

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

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

10.3 RegimeChangeDetected

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

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

10.4 ModeCollapseWarning

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

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

10.5 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.

10.5.1 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
24    return new_state, result
```

10.5.2 Dinámica Temporal

Ejemplo: Con `grace_period_steps=20`:

Interpretación:

1. En $t = 1$: Cambio genuino detectado. Se resetean pesos, inicia contador `grace_period_counter=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

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.

10.5.3 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.

10.5.4 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.

11 Manejo de Errores y Excepciones

11.1 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.

11.2 Alertas Específicas Avanzadas

- `ModeDegradationAlert`: Se emite cuando H_{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.

11.3 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 forensics 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

```

12 Detección de Mode Collapse en DGM

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

12.1 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`.

12.2 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.

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

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

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

donde δ 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 relajada 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.

14 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 σ_t puede explotar en múltiplos de 5 – 20× 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:

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

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         Orquestador 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:  # Siguientes 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            )
20
30            result = predictor.predict_next(observation)
31            depths_observed.append(result.metadata["signature_depth"])
32            latencies_observed.append(result.metadata["latency_ms"])
33
34            time.sleep(interval_ns / 1e9)  # Simular tick-rate
35
36        # Assertions
37        assert depths_observed[50] == 5, "Debe estar en M=5 durante régimen normal"
38        assert depths_observed[300] == 3, "Debe degradar a M=3 durante aceleración"
39        assert depths_observed[700] == 2, "Debe degradar a M=2 durante crisis"
40
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.