

# 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>Persistencia (Atomic Snapshotting)</b>	<b>6</b>
<b>6</b>	<b>I/O Asíncrono para Snapshots (Non-Blocking)</b>	<b>7</b>
<b>7</b>	<b>Ajuste Adaptativo del Umbral CUSUM</b>	<b>11</b>
7.1	Fórmula de Ajuste . . . . .	11
7.2	Interpretación de Curtosis . . . . .	11
<b>8</b>	<b>Periodo de Gracia (Ventana Refractaria) Post-Cambio de Régimen</b>	<b>11</b>
8.1	Motivación . . . . .	11
8.2	Solución: Grace Period (Refractario) . . . . .	12
8.3	Algoritmo de Implementación . . . . .	12
8.4	Parámetros Sugeridos . . . . .	13
8.5	Diagnóstico y Telemetría . . . . .	13
<b>9</b>	<b>Flags de Operación y Recuperación</b>	<b>13</b>
9.1	DegradedInferenceMode . . . . .	13
9.2	EmergencyMode . . . . .	13
9.3	RegimeChangeDetected . . . . .	13
9.4	ModeCollapseWarning . . . . .	13
9.5	Período de Gracia (Refractario) Post-Cambio de Régimen . . . . .	14
9.5.1	Implementación . . . . .	14
9.5.2	Dinámica Temporal . . . . .	14
9.5.3	Parámetros Recomendados . . . . .	15
9.5.4	Ventajas . . . . .	15
<b>10</b>	<b>Manejo de Errores y Excepciones</b>	<b>15</b>
10.1	Excepciones Estándar . . . . .	15
10.2	Alertas Específicas Avanzadas . . . . .	15
10.3	Ejemplo de Logging en Producción . . . . .	16

<b>11 Detección de Mode Collapse en DGM</b>	<b>17</b>
11.1 Criterio de Detección . . . . .	17
11.2 Acción Correctiva . . . . .	17
<b>12 Determinismo de Punto Flotante (Bit-Exact Reproducibility)</b>	<b>17</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 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 ( $y_t, y_{target}, \tau$ )

```
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_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 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
70             features."
71         )
72
73     self._state = self._deserialize_jax_state(payload)

```

## 6 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
37            os.replace(temp_filepath, filepath)
38
39            return {
40                'status': 'success',
41                'filepath': filepath,
42                'filesize_bytes': len(data_bytes),
43                'checksum': checksum,
44                'timestamp': time.time()
45            }
46        except Exception as e:
47            return {
48                'status': 'error',
49                'filepath': filepath,
50                'error': str(e),
51                'timestamp': time.time()
52            }
53
54    def save_snapshot_nonblocking(self, filepath: str) -> concurrent.futures.Future:
55        """
56            Exporta estado a snapshot sin bloquear el hilo de inferencia.
57
58            Retorna un Future<dict>. El llamante puede:
59            - Ignorarlo (fire-and-forget): permitir que escriba en background
60            - Esperar con .result(timeout=N): bloquear solo si es necesario (monitoreo)
61
62            Arquitectura:
63            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.modeCollapse_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 asincrónico 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 asincrónico 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']} "
141                                 f"({result['filesize_bytes']} bytes)")
142                         else:
143                             print(f"[WARNING] Snapshot falló: {result['error']}")
144                     except Exception as e:
145                         print(f"[ERROR] Future exception: {e}")
146
147     def graceful_shutdown(self, timeout_seconds=10):
148         """
149             Aguarda a que todos los snapshots pendientes se completen antes de cerrar.
150         """
151         print("[INFO] Aguardando snapshots pendientes...")
152
153         # Esperar a que se completen (con timeout)
154         concurrent.futures.wait(
155             [self.pending_snapshot_future] if self.pending_snapshot_future else [],
156             timeout=timeout_seconds
157         )
158
159         # Cerrar pool
160         self.io_executor.shutdown(wait=True)
161         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)
<b>Latencia observed por predictor</b>	<b>0.1–0.5</b>	<b>Principal</b>

Sin I/O asíncrono: latencia observada  $\sim$  7–65 ms Con I/O asíncrono: latencia observada  $\sim$  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  $\leq$  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)

```

## 7 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 régímenes con colas pesadas.

### 7.1 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

### 7.2 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 régímenes de alta curtosis mientras mantiene sensibilidad a cambios estructurales genuinos.

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

### 8.1 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.

## 8.2 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

## 8.3 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 líquidos o con gaps	Minimizar oscilaciones patológicas

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

## 8.4 Parámetros Sugeridos

## 8.5 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.

## 9 Flags de Operación y Recuperación

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

### 9.1 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  $\rho$  en último valor válido (modo inercial)
3. Predicciones continúan generándose pero con confianza degradada
4. Riesgo NO está siendo optimizado geométricamente

**Recuperación con histéresis:**

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

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

Se emite NormalOperationRestoredEvent al recuperar.

### 9.2 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.

### 9.3 RegimeChangeDetected

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

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

### 9.4 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.

## 9.5 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.

### 9.5.1 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 periodo 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 periodo 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

```

### 9.5.2 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.

- 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.

### 9.5.3 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.

### 9.5.4 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.

## 10 Manejo de Errores y Excepciones

### 10.1 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.

### 10.2 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). Señala al ejecutor que puede retomar operación normal.

### 10.3 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

```

## 11 Detección de Mode Collapse en DGM

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

### 11.1 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`.

### 11.2 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.

## 12 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:

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

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

donde  $\delta$  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_PLATFROMS'] = '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_PLATFROMS'] = '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.