Especificación de API Python - Universal Predictor

Ingeniería de Software

February 18, 2026

Contents

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)

```python
from dataclasses import dataclass
from typing import Optional
from jaxtyping import Float, Array, Bool

@dataclass(frozen=True)
class PredictorConfig:
    """Vector de Hiperparámetros Lambda."""
    schema_version: str = "1.0"   # Versionado de snapshots (evita incompatibilidades)
    epsilon: float = 1e-3         # Regularización Entrópica (Sinkhorn)
    learning_rate: float = 0.01   # Tasa de Aprendizaje JKO
    log_sig_depth: int = 3        # Profundidad de Firma (Kernel D)
    wtmm_buffer_size: int = 128   # Memoria WTMM (N_buf)
    besov_cone_c: float = 1.5     # Cono de Influencia de Besov
    holder_threshold: float = 0.4 # Umbral Circuit Breaker (H_min)
    cusum_h: float = 5.0          # Umbral Drift (h)
    cusum_k: float = 0.5          # Slack (k)
    grace_period_steps: int = 20  # Período refractario post-cambio régimen (silencia CUSUM)
    volatility_alpha: float = 0.1 # Decaimiento EWMA de Varianza

    # Política de Abandono y Anti-Aliasing
    staleness_ttl_ns: int = 500_000_000         # TTL Latencia (500ms)
    besov_nyquist_interval_ns: int = 100_000_000 # Límite Nyquist (100ms) para estabilidad WTMM
    inference_recovery_hysteresis: float = 0.8  # Factor histéresis para recuperación de modo degradado
```

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

```python
@dataclass(frozen=True)
class MarketObservation:
    price: Float[Array, "1"]      # y_t (Normalizado o Absoluto)
    target: Float[Array, "1"]     # y_target (Generalmente price actual)
    timestamp_ns: int             # Unix Epoch (Nanosegundos)

    def validate_domain(self, sigma_bound: float = 20.0, sigma_val: float = 1.0) -> bool:
        """Detección de Outliers Catastróficos (> 20 sigma)."""
        return abs(self.price) <= (sigma_bound * sigma_val)
```

2.3 Salida del Sistema

```python
@dataclass(frozen=True)
class PredictionResult:
    predicted_next: Float[Array, "1"]   # y_{t+1} (Espacio Z-Score)

    # Telemetría de Estado (S_risk)
    holder_exponent: Float[Array, "1"]  # H_t
    cusum_drift: Float[Array, "1"]      # G^+
    distance_to_collapse: Float[Array, "1"] # h - G^+
    free_energy: Float[Array, "1"]      # F (Energía JKO)

    # Telemetría Avanzada (Nuevas Adiciones)
    kurtosis: Float[Array, "1"]         #  _t - Curtosis empírica de errores
```

```
    dgm_entropy: Float[Array, "1"]       # H_DGM - Entropía del predictor DGM (NaN si inactivo)
    adaptive_threshold: Float[Array, "1"] # h_t - Umbral CUSUM adaptativo

    # Estado del Orquestador
    weights: Float[Array, "4"]           # [rho_A, rho_B, rho_C, rho_D] (Simplex)

    # Flags de Salud y Control (Explícitos)
    sinkhorn_converged: Bool[Array, "1"] # Convergencia JKO
    degraded_inference_mode: bool        # TTL violation (congelamiento de pesos)
    emergency_mode: bool                 # H_t < H_min (singularidad crítica)
    regime_change_detected: bool         # CUSUM alarm (G+ > h_t)
    mode_collapse_warning: bool          # H_DGM <  ·H[g] (colapso DGM)

    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.

```python
class FunctionalPredictor:
    """
    Implementación Stateless para JAX Core.
    Permite escalar a miles de predictores compartiendo la misma estructura computacional.
    """

    def __init__(self, config: PredictorConfig):
        # Compilación JIT única para todos los activos
        # Habilita vectorización automática (vmap) sobre la dimensión del batch (activos)
        self.config = config
        self._core_step = self._core_update_step
        self._jit_update = jax.jit(self._core_step)
        self._vmap_update = jax.jit(jax.vmap(self._core_step, in_axes=(0, 0, 0, 0)))

    def init_state(self):
        """Genera un estado cero inicial (cold state structure)."""
        return self._initialize_state_structure()

    def step(self, state, obs: MarketObservation) -> tuple[object, PredictionResult]:
        """
        Transición de Estado Pura: (S_t, Obs_t) -> (S_{t+1}, Pred_{t+1})
        """
        # 1. Validaciones (Outlier, Staleness, Nyquist) logic idéntica a UniversalPredictor
        # ... logic for freeze_weights flag calculation ...

        # 2. Ejecución Kernel JAX
        # Zero-Copy: La actualización de búferes ocurre dentro de XLA (dynamic_update_slice)
        new_state, raw_result = self._jit_update(
            state,  # Estado inyectado explícitamente desde Redis/Memoria
            obs.price,
            obs.target,
            freeze_weights=should_freeze
        )

        # 3. Mapeo de Resultados
        result = PredictionResult(
            predicted_next=raw_result.y_next,
            # ... resto de campos ...
        )

        return new_state, result

```

```
    def step_batch(self, states, obs_batch: MarketObservation):
        """
        Procesamiento vectorizado para N activos simultáneos.
        Utiliza vmap para paralelizar la inferencia y actualización.
        """
        # ... logic for batch flags ...
        new_states, results = self._vmap_update(states, obs_batch.price, obs_batch.target, freeze_flags)
        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

```
class UniversalPredictor:
    def __init__(self, config: PredictorConfig):
        """
        Inicializa el grafo de cómputo JAX (XLA JIT compilation).
        Asigna memoria estática para los búferes en el dispositivo (VRAM).
        El estado interno (self._state) contiene los `jnp.array` persistentes (rolling buffers)
        que se actualizarán mediante operaciones funcionales (jnp.roll, lax.dynamic_update)
        para eliminar la latencia de transferencia de memoria (Zero-Copy).
        """
        self.config = config
        self._state = self._initialize_state() # Estado interno JAX (resident un GPU)
        self._jit_update = jax.jit(self._core_update_step)
        self._last_timestamp_ns = 0 # Para cálculo de frecuencia

    def fit_history(self, history: list[float]) -> bool:
        """
        Bootstrapping inicial (Protocolo de Cold Start).
        Procesa el lote histórico para estabilizar los pesos JKO y llenar los búferes.
        Requiere un mínimo de N_buf muestras.

        Returns:
            bool: True si el sistema alcanzó convergencia estable (Sinkhorn + CUSUM).
        Raises:
            ValueError: Si el historial es insuficiente (< wtmm_buffer_size).
            RuntimeError: Si el sistema diverge tras el calentamiento.
        """
        if len(history) < self.config.wtmm_buffer_size:
            raise ValueError(f"Historial insuficiente. Requerido: {self.config.wtmm_buffer_size}")

        # Ejecución batch acelerada (jax.lax.scan) para calentar el estado
        # Simula el paso del tiempo para llenar colas y estabilizar gradientes
        self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))

        # Validación de Convergencia
        is_converged = final_metrics.sinkhorn_converged
        is_stable = final_metrics.cusum_drift < self.config.cusum_h

        if not (is_converged and is_stable):
             logger.warning("Cold Start finalizado sin convergencia estable.")
             return False

        return True
```

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

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

```
        """
        # 1. Validación de Dominio (Outlier Check)
        if not obs.validate_domain():
            logger.error("Outlier Catastrófico detectado. Ignorando tick.")
            return self._last_valid_result # Mantiene inercia

        # 2. Check de Abandono (Staleness) y Frecuencia (Anti-Aliasing)
        current_time = time.time_ns()
        latency = current_time - obs.timestamp_ns
        is_stale = latency > self.config.staleness_ttl_ns

        # Validación de Frecuencia Nyquist (WTMM Stability)
        dt_arrival = obs.timestamp_ns - self._last_timestamp_ns
        is_sparse = (self._last_timestamp_ns > 0) and (dt_arrival > self.config.
    besov_nyquist_interval_ns)

        if is_sparse:
             logger.warning(f"FrequencyWarning: Event interval {dt_arrival}ns > Nyquist limit.
     WTMM spectrum might alias.")

        self._last_timestamp_ns = obs.timestamp_ns

        # 3. Actualización Core (JAX) - Zero-Copy State Management
        # IMPORTANTE: El buffer de señal reside en GPU/TPU (self._state.signal_buffer).
        # La actualización se realiza "in-place" funcionalmente usando jax.lax.
    dynamic_update_slice
        # o jnp.roll dentro del kernel compilado para evitar transferencias CPU <-> VRAM.
        # Si hay staleness o sparsity excesiva, se congelan pesos para no degradar la geometrí
    a.
        should_freeze = is_stale or is_sparse

        new_state, result_data = self._jit_update(
            self._state,
            obs.price,
            obs.target,
            freeze_weights=should_freeze,
            # No se pasa history_buffer explícitamente, ya vive en _state
        )

        self._state = new_state

        # 4. Empaquetado de Resultados
        return PredictionResult(
            predicted_next=result_data.y_next,
            holder_exponent=result_data.H_t,
            sinkhorn_converged=result_data.converged,
            is_stable=not (is_stale or is_sparse),
            # ... mapeo resto de campos
        )
```

5 Prevención de Fragmentación de VRAM (JAX Memory Management)

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

Escenario:

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

Solución: Control Granular de Asignación de VRAM

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

```python
import os

# PASO 1: Limitar asignación inicial de VRAM
# Por defecto JAX asigna 90% → reservar solo 70% para dejar margen
os.environ['XLA_PYTHON_CLIENT_MEM_FRACTION'] = '0.7'

# PASO 2: Usar allocador "platform" para liberar dinámicamente si Python GC lo exige
# Opciones:
#   'platform' (recomendado): Libera memoria al solicitar el SO
#   'bfc'     (default): Caché de bloques fija (menos flexible)
os.environ['XLA_PYTHON_CLIENT_ALLOCATOR'] = 'platform'

# PASO 3: Habilitar protección contra fragmentación
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'  # Permite crecimiento incremental

# Ahora importar JAX (después de fijar variables de entorno)
import jax
import jax.numpy as jnp

print(f"VRAM allocation: {jax.devices()}")
print(f"XLA allocator: platform (dynamic)")
```

Implicaciones de Configuración:

Variable	Default	Recomendado	Efecto
XLA_PYTHON_CLIENT_MEM_FRACTION	0.9	0.7	Deja 30% libre para I/O, SO, buffer
XLA_PYTHON_CLIENT_ALLOCATOR	bfc	platform	Libera dinamicamente 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:

```python
class VRAM_PooledAllocator:
    def __init__(self, device_memory_budget_gb: float = 28.0, pool_size: int = 100):
        """
        Crea un pool de tensores pre-asignados para reducir fragmentación.
        """
        self.device_budget_bytes = device_memory_budget_gb * 1e9
        self.pool = []
        self.available = []

        # Pre-asignar tensores comunes (ej. buffers de Sinkhorn)
        for i in range(pool_size):
            tensor = jnp.zeros((1024, 1024), dtype=jnp.float32)  # ~4MB
            self.pool.append(tensor)
            self.available.append(True)

    def acquire_tensor(self, shape, dtype=jnp.float32):
        """Obtiene tensor del pool sin fragmentar VRAM."""
        for idx, available in enumerate(self.available):
            if available:
                self.available[idx] = False
                return self.pool[idx]

        # Si no hay disponible, crear temporalmente
        return jax.device_put(jnp.zeros(shape, dtype=dtype))

    def release_tensor(self, idx):
```

```
        """Devuelve tensor al pool."""
        if idx < len(self.available):
            self.available[idx] = True

    def memory_utilization_percent(self):
        """Reporta fragmentación."""
        used = sum(1 for av in self.available if not av)
        return 100.0 * used / len(self.available)
```

Monitoreo de Fragmentación:

```
import psutil
import subprocess

def monitor_vram_fragmentation(interval_seconds=60):
    """
    Thread de monitoreo que reporta fragmentación de VRAM.
    """
    import time
    import threading

    def _monitor():
        while True:
            try:
                # Consultar nvidia-smi para obtener uso real
                result = subprocess.run(
                    ['nvidia-smi', '--query-gpu=memory.used,memory.total',
                     '--format=csv,nounits,noheader'],
                    capture_output=True, text=True, timeout=5
                )

                if result.returncode == 0:
                    used, total = map(float, result.stdout.strip().split(','))
                    utilization = 100.0 * used / total

                    if utilization > 0.95:
                        print(f"[WARNING] VRAM near saturation: {utilization:.1f}%")
                    elif utilization > 0.85:
                        print(f"[INFO] VRAM utilization: {utilization:.1f}% (elevated)")

                time.sleep(interval_seconds)
            except Exception as e:
                print(f"[ERROR] VRAM monitoring failed: {e}")
                break

    thread = threading.Thread(target=_monitor, daemon=True)
    thread.start()
```

Configuración de Despliegue Recomendada:

```
#!/bin/bash
# deployment/run_predictor.sh

# Variables de entorno CRÍTICAS para producción
export XLA_PYTHON_CLIENT_MEM_FRACTION=0.7
export XLA_PYTHON_CLIENT_ALLOCATOR=platform
export TF_FORCE_GPU_ALLOW_GROWTH=true

# Logs de configuración
echo "[INFO] XLA VRAM Fraction: 0.7 (28/40 GB en A100)"
echo "[INFO] Allocator: platform (dynamic)"
echo "[INFO] GPU growth: enabled"

# Ejecutar predictor
python3 -u predictor_service.py \
    --config config.yaml \
    --device gpu \
    --pool-size 100 \
    --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.

```python
import hashlib
import msgpack

    def save_snapshot(self, filepath: str):
        """
        Exporta el estado interno Sigma_t a formato binario (MessagePack).
        Incluye Checksum SHA-256 al final del archivo.
        """
        # Serialización de tensores JAX a bytes
        state_dict = self._serialize_jax_state(self._state)

        # Segmentación Modular (K-Blocks)
        # IMPORTANTE: Incluir versionado de schema para evitar errores al cargar
        # snapshots generados con versiones antiguas (cambios en profundidad de firma, etc.)
        payload = {
            "schema_version": self.config.schema_version,  # Versionado seguro
            "timestamp": time.time_ns(),
            "config": asdict(self.config),
            "global": state_dict["global"], # rho, G+, ema
            "telemetry": {
                "kurtosis": float(self._state.kurtosis),
                "dgm_entropy": float(self._state.dgm_entropy),
                "adaptive_threshold": float(self._state.h_adaptive)
            },
            "flags": {
                "degraded_inference": bool(self._state.degraded_mode),
                "emergency": bool(self._state.emergency_mode),
                "regime_change": bool(self._state.regime_changed),
                "mode_collapse": bool(self._state.mode_collapse_warning)
            },
            "kernels": {
                "A": state_dict["kernel_a"],
                "B": state_dict["kernel_b"],
                "C": state_dict["kernel_c"],
                "D": state_dict["kernel_d"]
            }
        }

        data_bytes = msgpack.packb(payload)
        checksum = hashlib.sha256(data_bytes).hexdigest()

        with open(filepath, "wb") as f:
            f.write(data_bytes)
            f.write(checksum.encode('utf-8')) # Append hash

    def load_snapshot(self, filepath: str):
        """
        Carga estado. Valida SHA-256 y schema_version antes de deserializar.
        Lanza ValueError si falla la validación o schema incompatible.
        """
        with open(filepath, "rb") as f:
            content = f.read()

        data_bytes = content[:-64] # Todo menos los últimos 64 bytes (SHA256 hex)
        stored_checksum = content[-64:].decode('utf-8')

        computed = hashlib.sha256(data_bytes).hexdigest()
        if computed != stored_checksum:
            raise ValueError("Snapshot corrupto: Checksum mismatch.")

        payload = msgpack.unpackb(data_bytes)

        # Validar schema_version para detectar incompatibilidades
        loaded_schema = payload.get('schema_version', 'unknown')
```

```
        if loaded_schema != self.config.schema_version:
            raise ValueError(
                f"Schema version mismatch: snapshot={loaded_schema}, "
                f"current={self.config.schema_version}. "
                f"Cannot load snapshot generated with incompatible kernel depths or signature features."
            )

        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:

```
import concurrent.futures
import hashlib
import msgpack
import threading
import time

class UniversalPredictor_AsyncIO:
    def __init__(self, n_worker_threads=2):
        # Pool de threads dedicados a I/O (no interfieren con inferencia)
        self.io_executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=n_worker_threads,
            thread_name_prefix="snapshot_io_"
        )

        # Futuro del snapshot en vuelo para monitoreo (opcional)
        self.pending_snapshot_future = None
        self.snapshot_lock = threading.Lock()

    def _compute_and_save_async(self, filepath: str, data_bytes: bytes):
        """
        Ejecuta en thread pool: calcula checksum y escribe a disco.
        No bloquea el hilo principal.
        """
        checksum = hashlib.sha256(data_bytes).hexdigest()

        # Escritura atómica (write + rename) para evitar estado intermedio
        temp_filepath = filepath + ".tmp"

        try:
            with open(temp_filepath, "wb") as f:
                f.write(data_bytes)
                f.write(checksum.encode('utf-8'))

            # Rename atómico (POSIX-compliant)
            import os
```

```
            os.replace(temp_filepath, filepath)

            return {
                'status': 'success',
                'filepath': filepath,
                'filesize_bytes': len(data_bytes),
                'checksum': checksum,
                'timestamp': time.time()
            }
        except Exception as e:
            return {
                'status': 'error',
                'filepath': filepath,
                'error': str(e),
                'timestamp': time.time()
            }

    def save_snapshot_nonblocking(self, filepath: str) -> concurrent.futures.Future:
        """
        Exporta estado a snapshot sin bloquear el hilo de inferencia.

        Retorna un Future<dict>. El llamante puede:
        - Ignorarlo (fire-and-forget): permitir que escriba en background
        - Esperar con .result(timeout=N): bloquear solo si es necesario (monitoreo)

        Arquitectura:
        1. Main thread: Serialización MessagePack (rápido: microsegundos)
        2. Main thread: Retorna control inmediatamente
        3. Worker thread: Cálculo SHA-256 + escritura a disco (en background)
        """

        # Paso 1: Serialización (en el hilo principal, muy rápido)
        state_dict = self._serialize_jax_state(self._state)

        payload = {
            "schema_version": self.config.schema_version,
            "timestamp": time.time_ns(),
            "config": asdict(self.config),
            "global": state_dict["global"],
            "telemetry": {
                "kurtosis": float(self._state.kurtosis),
                "dgm_entropy": float(self._state.dgm_entropy),
                "adaptive_threshold": float(self._state.h_adaptive)
            },
            "flags": {
                "degraded_inference": bool(self._state.degraded_mode),
                "emergency": bool(self._state.emergency_mode),
                "regime_change": bool(self._state.regime_changed),
                "mode_collapse": bool(self._state.mode_collapse_warning)
            },
            "kernels": {
                "A": state_dict["kernel_a"],
                "B": state_dict["kernel_b"],
                "C": state_dict["kernel_c"],
                "D": state_dict["kernel_d"]
            }
        }

        data_bytes = msgpack.packb(payload)

        # Paso 2: Delegar I/O a thread pool (no-bloqueante)
        future = self.io_executor.submit(
            self._compute_and_save_async,
            filepath,
            data_bytes
        )

        # Mantener referencia al Future para debugging (opcional)
        with self.snapshot_lock:
            self.pending_snapshot_future = future

        return future

```

```python
    def predict_step_with_async_checkpoint(self, x_t: jnp.ndarray) -> Tuple[jnp.ndarray,
    Optional[concurrent.futures.Future]]:
        """
        Paso de predicción con snapshot asíncrono periódico.

        El snapshot se dispara cada N pasos pero NO interfiere con latencia de inferencia.
        """
        # Predicción principal (hot path)
        prediction = self._predict_step_core(x_t)

        # Checkpoint asíncrono si es el momento
        snapshot_future = None
        if self.step_counter % self.checkpoint_interval == 0:
            checkpoint_path = f"{self.checkpoint_dir}/snapshot_step_{self.step_counter}.
    msgpack"
            snapshot_future = self.save_snapshot_nonblocking(checkpoint_path)
            # El hilo retorna SIN esperar a que el snapshot se escriba a disco

        self.step_counter += 1
        return prediction, snapshot_future

    def monitor_snapshot_queue(self):
        """
        Thread de monitoreo (opcional) que reporta el estado de snapshots en vuelo.
        Ejecutado en otro thread para no interferir.
        """
        while not self.shutdown_event.wait(timeout=5.0):
            with self.snapshot_lock:
                if self.pending_snapshot_future is not None:
                    if self.pending_snapshot_future.done():
                        try:
                            result = self.pending_snapshot_future.result()
                            if result['status'] == 'success':
                                print(f"[INFO] Snapshot guardado: {result['filepath']} "
                                      f"({result['filesize_bytes']} bytes)")
                            else:
                                print(f"[WARNING] Snapshot falló: {result['error']}")
                        except Exception as e:
                            print(f"[ERROR] Future exception: {e}")

    def graceful_shutdown(self, timeout_seconds=10):
        """
        Aguarda a que todos los snapshots pendientes se completen antes de cerrar.
        """
        print("[INFO] Aguardando snapshots pendientes...")

        # Esperar a que se completen (con timeout)
        concurrent.futures.wait(
            [self.pending_snapshot_future] if self.pending_snapshot_future else [],
            timeout=timeout_seconds
        )

        # Cerrar pool
        self.io_executor.shutdown(wait=True)
        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 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 retornado permite que el snapshot se escriba en background sin interferencia.

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

Configuración Recomendada:

```
predictor = UniversalPredictor_AsyncIO(
    n_worker_threads=2,  # Típicamente 1-2 threads suficientes
    checkpoint_interval=1000  # Cada 1000 pasos (~1 segundo en latencia 1ms)
)

# En el loop de trading/predicción:
for x_t in market_stream:
    prediction, snapshot_future = predictor.predict_step_with_async_checkpoint(x_t)

    # USA prediction inmediatamente
    # El snapshot se escribirá en background
    if snapshot_future is not None:
        # Opcional: esperar solo en situaciones críticas (ej. antes de shutdown)
        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 regímenes 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 regímenes 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):

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

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

9.3 Algoritmo de Implementación

```python
class CUSUMState:
    def __init__(self, grace_period_steps=20):
        self.g_plus = 0.0
        self.g_minus = 0.0
        self.error_sq_ema = 0.0
        self.steps_since_regime_change = 0
        self.grace_period = grace_period_steps

    def step(self, error, sigma_t, kurtosis):
        """Avanza el estado CUSUM con silenciamiento refractario."""
        # Incrementar contador desde último cambio
        self.steps_since_regime_change += 1

        # Calcular estadística (siempre)
        k = self.config.cusum_k
        h_adaptive = k * sigma_t * (1 + np.log(max(kurtosis, 1.0) / 3.0))

        s_standardized = np.abs(error) / sigma_t
        s_centered = s_standardized - 1.0

        self.g_plus = max(0.0, self.g_plus + s_centered - k)

        # Lógica de alarma CON GRACIA
        is_in_grace_period = (
```

```
            self.steps_since_regime_change < self.grace_period
        )

        if is_in_grace_period:
            # Silenciar: no emitir alarma
            alarm = False
        else:
            # Normal: comparar con umbral
            alarm = (self.g_plus > h_adaptive)

        return alarm, self.g_plus, h_adaptive

    def reset_on_regime_change(self):
        """Al detectar cambio, iniciar período de gracia."""
        self.g_plus = 0.0
        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 ilíquidos 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 señalizan 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:

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

donde $h_{\text{hyst}} =$ `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 $w_D \to 1.0$ (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: $H_{\text{DGM}} < \gamma \cdot H[g]$ durante > 10 pasos consecutivos (solo relevante si $\rho_B > 0.05$)
Acción correctiva: Reducir $\rho_B \to 0$ 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:

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

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

```
def _core_update_step(state, price, target, freeze_weights=False):
    # ... paso de identificación (SIA, WTMM) ...

    # Calcular CUSUM normalmente
    raw_alarm = self._check_regime_change_with_kurtosis(error)

    # Aplicar período de gracia: silenciar falsa alarma si dentro de gracia
    if state.grace_period_counter > 0:
        raw_alarm = False  # Suprimir detección durante refractario
        state.grace_period_counter -= 1
        state.regime_change_locked = True
    else:
        state.regime_change_locked = False

    # Si se detecta cambio FUERA del período de gracia, resetear contador
    if raw_alarm and not state.regime_change_locked:
        state.grace_period_counter = self.config.grace_period_steps
        # Reiniciar pesos a uniforme, resetear acumuladores
        weights = jnp.ones(4) / 4.0
        cusum_state['g_plus'] = 0.0

    # ... resto de la lógica ...
    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

```python
import logging
import os
from datetime import datetime

def save_emergency_dump(predictor, result, asset_id: str):
    """
    Guarda un "Dump de Depuración" completo cuando se activa EmergencyMode.
    Incluye: estado de pesos, buffer de señales, historial de telemetría.
    """
    dump_dir = os.path.expanduser("~/.predictor_emergency_dumps")
    os.makedirs(dump_dir, exist_ok=True)

    timestamp = datetime.now().isoformat()
    dump_file = f"{dump_dir}/{asset_id}_emergency_{timestamp}.msgpack"

    debug_payload = {
        "emergency_timestamp": timestamp,
        "asset_id": asset_id,
        "holder_exponent": float(result.holder_exponent),
        "weights": [float(w) for w in result.weights],
        "signal_buffer": predictor._state.signal_circular_buffer.tolist(),
        "regime_history": predictor._state.cusum_history.tolist(),
        "telemetry_snapshot": {
            "kurtosis": float(result.kurtosis),
            "dgm_entropy": float(result.dgm_entropy),
            "adaptive_threshold": float(result.adaptive_threshold),
            "distance_to_collapse": float(result.distance_to_collapse)
        },
        "flags_at_emergency": {
            "degraded_inference": bool(result.degraded_inference_mode),
            "regime_change": bool(result.regime_change_detected),
            "mode_collapse": bool(result.mode_collapse_warning)
        }
    }

    with open(dump_file, "wb") as f:
        msgpack.packb(debug_payload, file=f)

    logging.critical(f"Emergency dump saved to {dump_file} for forensius analysis")

def process_prediction(predictor, obs):
    result = predictor.step(obs)
    asset_id = obs.asset_id if hasattr(obs, 'asset_id') else "unknown"

    # Flags críticos
    if result.degraded_inference_mode:
        logging.warning(
            "DEGRADED MODE: TTL exceeded. Weights frozen. "
            "Consider reducing position size."
        )

    if result.emergency_mode:
        logging.critical(
            f"EMERGENCY: Singularity detected (H={result.holder_exponent:.3f}). "
            "Forcing Kernel D with Huber loss."
        )
        # Guardar dump automáticamente para análisis post-mortem
```

```
        save_emergency_dump(predictor, result, asset_id)

    if result.mode_collapse_warning:
        logging.error(
            f"MODE COLLAPSE: DGM entropy below threshold. "
            f"H_DGM = {result.dgm_entropy:.3f}. "
            "Reducing rho_B -> 0."
        )

    if result.kurtosis > 20.0:
        logging.warning(
            f"KURTOSIS OUTLIER: kappa = {result.kurtosis:.2f} > 20. "
            "Residual model may be invalid."
        )

    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:

$$\rho_B \to 0$$

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

Nota Teórica: Una solución colapsada tiene $H[V_\theta] \to -\infty$ (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 $\to$ error de redondeo ϵ_{CPU}
- En GPU: suma paralela con diferentes agrupaciones $\to$ error $\epsilon_{\text{GPU}} \neq \epsilon_{\text{CPU}}$
- En FPGA: precisión de punto flotante custom $\to$ 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:

```
import os
import jax
import jax.numpy as jnp

# PASO 1: Configurar variables de entorno ANTES de cualquier operación JAX
os.environ['XLA_FLAGS'] = '--xla_cpu_use_cross_replica_callbacks=false'
os.environ['JAX_DETERMINISTIC_REDUCTIONS'] = '1'  # Force deterministic reductions
os.environ['JAX_TRACEBACK_FILTERING'] = 'off'     # Completo traceback para debugging

# Alternativas según backend:
# Para GPU (CUDA):
# os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
# os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':16:8'  # Fijo cuBLAS workspace

# Para forzar CPU (deshabilitar GPU):
# os.environ['JAX_PLATFORMS'] = 'cpu'

# PASO 2: Fijar semillas globales de PRNG
import numpy as np

RANDOM_SEED = 42  # Determinista en todos los backends

# Semilla NumPy
np.random.seed(RANDOM_SEED)

# Semilla JAX (RNG de clave)
from jax import random
jax.config.update('jax_default_prng_impl', 'threefry2x32')  # Determinista entre backends
key = random.PRNGKey(RANDOM_SEED)

# PASO 3: Importar y configurar JAX después de las variables de entorno
jax.config.update('jax_enable_x64', True)  # float64 para mayor precisión

# PASO 4: Verificar que se forzó determinismo
print(f"XLA Backend: {jax.devices()}")
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):

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

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

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

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

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):

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

```

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

2. **GPU Determinista**:

```
os.environ['JAX_PLATFORMS'] = 'gpu'
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':16:8'
# Con las mismas semillas y configuración XLA
result_gpu = predictor.predict_step(x_t, key)

assert jnp.allclose(result_cpu, result_gpu, atol=1e-7, rtol=1e-6)

```

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):

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

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

```

Procedimiento de Validación en CI/CD:

```
#!/bin/bash
# File: tests/determinism_test.sh

set -e

echo "[CPU] Running determinism test..."
export JAX_DETERMINISTIC_REDUCTIONS=1
export JAX_PLATFORMS=cpu
python tests/test_cpu_baseline.py

echo "[GPU] Running determinism test on GPU..."
export JAX_PLATFORMS=gpu
export CUDA_LAUNCH_BLOCKING=1
export CUBLAS_WORKSPACE_CONFIG=:16:8
python tests/test_gpu_consistency.py

echo "[FPGA Sim] Running determinism test with float32..."
python tests/test_fpga_sim.py

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.