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 Apagado Elegante (Graceful Shutdown) para Contenedores

Problema en Entornos Orquestados (Docker/Kubernetes):

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

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

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

Consecuencia de Apagado Abrupto (sin manejo de SIGTERM):

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

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

Solución: Captura de SIGTERM con Protocolo de Shutdown Ordenado

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

Implementación:

```python
import signal
import sys
import threading
import time
import logging
from typing import Optional

class UniversalPredictor_GracefulShutdown:
    def __init__(self, config: PredictorConfig):
        self.config = config
        self.predictor = UniversalPredictor_AsyncIO(config)

        # Control de ciclo de vida
        self.shutdown_requested = threading.Event()
        self.is_accepting_data = True
        self.input_buffer_lock = threading.Lock()
        self.residual_buffer = []  # Buffer de observaciones no procesadas

        # Registro de señales POSIX
        signal.signal(signal.SIGTERM, self._handle_sigterm)
        signal.signal(signal.SIGINT, self._handle_sigint)  # Ctrl+C en desarrollo

        # Logger para auditoría
        self.logger = logging.getLogger("predictor.shutdown")
        self.logger.info("[INIT] Graceful shutdown handler registered")

    def _handle_sigterm(self, signum, frame):
        """
        Manejador de SIGTERM (enviado por Kubernetes/Docker).
        Inicia protocolo de apagado elegante.
        """
        self.logger.warning(f"[SIGTERM] Received signal {signum}. Initiating graceful shutdown
    ...")
        self.shutdown_requested.set()

    def _handle_sigint(self, signum, frame):
        """
        Manejador de SIGINT (Ctrl+C local, desarrollo).
        """
        self.logger.warning(f"[SIGINT] Received signal {signum}. Initiating graceful shutdown
    ...")
        self.shutdown_requested.set()

    def accept_observation(self, obs: MarketObservation) -> Optional[PredictionResult]:
        """
        Método thread-safe para recibir observaciones.
        Rechaza nuevas entradas si shutdown está en curso.
        """
        if self.shutdown_requested.is_set() or not self.is_accepting_data:
            self.logger.warning(f"[REJECT] Observation rejected (shutdown in progress): {obs.
    timestamp_ns}")
            return None

        with self.input_buffer_lock:
            self.residual_buffer.append(obs)

        # Procesamiento inmediato (o delegación a cola)
        return self._process_observation(obs)

    def _process_observation(self, obs: MarketObservation) -> PredictionResult:
        """
        Procesa observación y emite predicción.
        """
        result = self.predictor.predict_next(obs)

        with self.input_buffer_lock:
            # Remover de buffer residual ya que se procesó
            if obs in self.residual_buffer:
                self.residual_buffer.remove(obs)

        return result

    def graceful_shutdown(self, timeout_seconds: int = 25):
```

```
        """
        Protocolo de apagado elegante (llamado automáticamente por SIGTERM).

        Pasos:
        1. Bloquear nuevas observaciones
        2. Procesar buffer residual
        3. Esperar snapshots asíncronos pendientes
        4. Ejecutar snapshot final
        5. Cerrar recursos (sockets, threads)
        6. Salir con código 0

        Args:
            timeout_seconds: Tiempo máximo antes de rendirse (debe ser < grace period de K8s)
        """
        shutdown_start = time.time()
        self.logger.info("                    ")
        self.logger.info("  GRACEFUL SHUTDOWN INITIATED")
        self.logger.info("                    ")

        # PASO 1: Bloquear nuevas entradas
        self.logger.info("[1/5] Blocking new observations...")
        self.is_accepting_data = False
        time.sleep(0.1)  # Dar tiempo a threads en vuelo para terminar
        self.logger.info("        Input blocked:  ")

        # PASO 2: Procesar buffer residual
        self.logger.info("[2/5] Processing residual buffer...")
        with self.input_buffer_lock:
            residual_count = len(self.residual_buffer)
            self.logger.info(f"        Residual observations: {residual_count}")

            for obs in list(self.residual_buffer):
                if time.time() - shutdown_start > timeout_seconds - 10:
                    self.logger.warning("        Timeout approaching, aborting residual
processing")
                    break

                try:
                    _ = self._process_observation(obs)
                except Exception as e:
                    self.logger.error(f"        Error processing residual: {e}")

            remaining = len(self.residual_buffer)
            if remaining > 0:
                self.logger.warning(f"        {remaining} observations dropped (timeout)")
            else:
                self.logger.info("        Residual buffer cleared:  ")

        # PASO 3: Esperar snapshots asíncronos pendientes
        self.logger.info("[3/5] Waiting for pending async snapshots...")
        pending_snapshot = self.predictor.pending_snapshot_future

        if pending_snapshot is not None and not pending_snapshot.done():
            try:
                remaining_time = max(1, timeout_seconds - (time.time() - shutdown_start))
                pending_snapshot.result(timeout=remaining_time)
                self.logger.info("        Async snapshot completed:  ")
            except concurrent.futures.TimeoutError:
                self.logger.error("        Async snapshot timed out (may be corrupted)")
            except Exception as e:
                self.logger.error(f"        Async snapshot failed: {e}")
        else:
            self.logger.info("        No pending snapshots:  ")

        # PASO 4: Snapshot final (síncrono, garantizado)
        self.logger.info("[4/5] Executing final snapshot (synchronous)...")
        try:
            final_snapshot_path = f"snapshots/shutdown_{int(time.time())}.pkl"
            self.predictor.save_snapshot(final_snapshot_path)
            self.logger.info(f"        Final snapshot saved: {final_snapshot_path}")
        except Exception as e:
            self.logger.error(f"        Final snapshot FAILED: {e}")
            # Continuar de todos modos, no queremos bloquear el shutdown
```

```
        # PASO 5: Cerrar recursos
        self.logger.info("[5/5] Closing resources...")
        try:
            # Cerrar ThreadPoolExecutor (si existe)
            if hasattr(self.predictor, 'io_executor'):
                self.predictor.io_executor.shutdown(wait=True, cancel_futures=False)
                self.logger.info("        ThreadPoolExecutor closed: ")

            # Cerrar sockets de mercado (si aplica)
            # self.market_socket.close()

            # Limpiar caché JAX (opcional)
            # jax.clear_caches()

        except Exception as e:
            self.logger.error(f"        Error closing resources: {e}")

        # Resumen final
        total_time = time.time() - shutdown_start
        self.logger.info("                    ")
        self.logger.info(f"  SHUTDOWN COMPLETED ({total_time:.2f}s)")
        self.logger.info("                    ")

        # Salir con código 0 (limpio)
        sys.exit(0)

    def run_forever(self):
        """
        Bucle principal del predictor.
        Monitorea shutdown_requested en cada iteración.
        """
        self.logger.info("[MAIN] Predictor running. Waiting for market data...")

        try:
            while not self.shutdown_requested.is_set():
                # Simular recepción de datos (en producción: socket.recv, kafka.poll, etc.)
                # observation = self.market_socket.recv()
                # result = self.accept_observation(observation)

                time.sleep(0.001)  # Evitar busy-wait

        except KeyboardInterrupt:
            # Capturado por _handle_sigint, pero por si acaso
            self.logger.warning("[MAIN] KeyboardInterrupt detected")

        # Shutdown solicitado
        self.graceful_shutdown(timeout_seconds=25)
```

Integración con Kubernetes ConfigMap:

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: universal-predictor
spec:
  replicas: 3
  template:
    spec:
      terminationGracePeriodSeconds: 30  # Tiempo para graceful_shutdown()
      containers:
      - name: predictor
        image: predictor:v1.0
        command: ["python", "main.py"]
        lifecycle:
          preStop:
            exec:
              # Kubernetes ejecuta esto ANTES de enviar SIGTERM
              # Opcional: Notificar load balancer que el pod está muriendo
              command: ["/bin/sh", "-c", "sleep 2"]
        resources:
          requests:
            memory: "16Gi"
```

```
            nvidia.com/gpu: 1
          limits:
            memory: "32Gi"
            nvidia.com/gpu: 1
```

Script Principal (main.py):

```python
#!/usr/bin/env python3
import logging
from stochastic_predictor import UniversalPredictor_GracefulShutdown, PredictorConfig

def main():
    # Configurar logging
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s [%(name)s] %(levelname)s: %(message)s',
        handlers=[
            logging.StreamHandler(),
            logging.FileHandler("/var/log/predictor.log")
        ]
    )

    logger = logging.getLogger("main")
    logger.info("Starting Universal Predictor...")

    # Cargar configuración
    config = PredictorConfig.from_json("config.json")

    # Inicializar predictor con graceful shutdown
    predictor = UniversalPredictor_GracefulShutdown(config)

    # Ejecutar (bloqueante hasta SIGTERM)
    predictor.run_forever()

    # NUNCA llega aquí (sys.exit(0) en graceful_shutdown)

if __name__ == "__main__":
    main()
```

Prueba de Apagado Elegante:

```python
def test_graceful_shutdown_protocol():
    """
    Simula apagado elegante con buffer residual y snapshot pendiente.
    """
    import signal
    import os
    import time

    config = PredictorConfig(checkpoint_interval=100)
    predictor = UniversalPredictor_GracefulShutdown(config)

    # Simular carga de trabajo
    for i in range(50):
        obs = MarketObservation(
            price=jnp.array([float(i)]),
            target=jnp.array([0.0]),
            timestamp_ns=time.time_ns()
        )
        predictor.accept_observation(obs)

    # Disparar snapshot asíncrono
    predictor.predictor.save_snapshot_nonblocking("test_snapshot.pkl")

    # Añadir observaciones al buffer residual
    for i in range(10):
        obs = MarketObservation(
            price=jnp.array([float(100 + i)]),
            target=jnp.array([0.0]),
            timestamp_ns=time.time_ns()
        )
        with predictor.input_buffer_lock:
            predictor.residual_buffer.append(obs)

    # Simular SIGTERM (en test, llamamos directamente al handler)
```

```python
    predictor._handle_sigterm(signal.SIGTERM, None)

    # Verificar que shutdown se ejecutó
    assert predictor.shutdown_requested.is_set(), "Shutdown not triggered"

    # Ejecutar shutdown manualmente (en producción es automático)
    try:
        predictor.graceful_shutdown(timeout_seconds=10)
    except SystemExit as e:
        assert e.code == 0, f"Exit code should be 0, got {e.code}"

    # Verificar que snapshot final existe
    final_snapshots = [f for f in os.listdir("snapshots") if f.startswith("shutdown_")]
    assert len(final_snapshots) > 0, "Final snapshot not created"

    print(" [] Graceful shutdown test PASSED")
    print(f"    Final snapshot: {final_snapshots[-1]}")
    print(f"    Residual buffer: cleared")
    print(f"    Async snapshots: completed")
```

Cronología de Shutdown (Típica):

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

Comparación: Shutdown Abrupto vs Elegante

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

Configuración de Timeouts (Recomendaciones):

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

Consideraciones de Producción:

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

4. **Persistencia Distribuida**: Guardar snapshots finales en almacenamiento compartido (NFS, S3) en lugar de volumen local del pod

5. **Logging Estructurado**: Todos los logs de shutdown deben ser JSON con campo `shutdown_phase` para análisis post-mortem

6. **Métricas de Shutdown**: Exponer contador Prometheus `predictor_graceful_shutdowns_total` y histogram `predictor_shutdown_duration_seconds`

Integración con Prometheus:

```python
from prometheus_client import Counter, Histogram

class UniversalPredictor_GracefulShutdown_Monitored:
    def __init__(self, config: PredictorConfig):
        # ... (inicialización base) ...

        # Métricas de shutdown
        self.shutdown_counter = Counter(
            'predictor_graceful_shutdowns_total',
            'Total number of graceful shutdowns executed'
        )
        self.shutdown_duration = Histogram(
            'predictor_shutdown_duration_seconds',
            'Time taken to complete graceful shutdown',
            buckets=[0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0]
        )
        self.residual_observations = Histogram(
            'predictor_shutdown_residual_observations',
            'Number of observations in buffer during shutdown',
            buckets=[0, 1, 5, 10, 50, 100, 500, 1000]
        )

    def graceful_shutdown(self, timeout_seconds: int = 25):
        self.shutdown_counter.inc()

        with self.shutdown_duration.time():
            # ... (código de shutdown) ...

            with self.input_buffer_lock:
                self.residual_observations.observe(len(self.residual_buffer))

            # ... (resto del shutdown) ...
```

Conclusión:

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

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

Con el protocolo implementado:

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

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

9 Ajuste Adaptativo del Umbral CUSUM

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

9.1 Fórmula de Ajuste

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

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

donde:

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

9.2 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.

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

10.1 Motivación

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

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

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

10.2 Solución: Grace Period (Refractario)

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

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

Durante el período de gracia:

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

Transcurrido el período:

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

10.3 Algoritmo de Implementación

```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
```

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

Table 2: Recomendaciones para grace_period_steps según régimen

10.4 Parámetros Sugeridos

10.5 Diagnóstico y Telemetría

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

- G^+ observable (si hubiera alarma)
- 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.

11 Flags de Operación y Recuperación

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

11.1 DegradedInferenceMode

Condición de activación:

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

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.

11.2 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.

11.3 RegimeChangeDetected

Condición: $G^+ > h_t$ (CUSUM detecta cambio de régimen)
Acción: Reinicio de entropía a distribución uniforme y reset de acumuladores.

11.4 ModeCollapseWarning

Condición: $H_{\text{DGM}} < \gamma \cdot H[g]$ durante > 10 pasos consecutivos (solo relevante si $\rho_B > 0.05$)
Acción correctiva: Reducir $\rho_B \to 0$ hasta re-entrenar red DGM.

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

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

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

11.5.1 Implementación

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

```python
@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:

```python
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
```

11.5.2 Dinámica Temporal

Ejemplo: Con `grace_period_steps=20`:

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

Table 3: Evolución del contador de gracia con detecciones repetidas

Interpretación:

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

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

11.5.3 Parámetros Recomendados

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

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

11.5.4 Ventajas

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

12 Manejo de Errores y Excepciones

12.1 Excepciones Estándar

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

12.2 Alertas Específicas Avanzadas

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

12.3 Ejemplo de Logging en Producción

```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
```

13 Detección de Mode Collapse en DGM

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

13.1 Criterio de Detección

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

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

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

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

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

13.2 Acción Correctiva

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

$$\rho_B \to 0$$

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.

14 Determinismo de Punto Flotante (Bit-Exact Reproducibility)

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

Impacto:

- En CPU: suma secuencial $\to$ 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:

```python
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):
```

```python
# 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):

```python
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**:

```python
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):

```python
# 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)
```

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

Procedimiento de Validación en CI/CD:

```
#!/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.

15 Load Shedding Dinámico (Poda Topológica en Cisne Negro)

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

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

Consecuencia sin Mitigación:

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

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

Solución: Load Shedding Adaptativo en Orquestador

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

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

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

Implementación: Dual JIT Graph Switching

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

```python
import jax
import jax.numpy as jnp
from collections import deque
from typing import Dict, Callable

class AdaptiveSignatureOrchestrator:
    """
    Orquestrador con Load Shedding para Rama D.
    Mantiene múltiples grafos JIT precompilados y selecciona dinámicamente.
    """
    def __init__(self, config: PredictorConfig):
        self.config = config
        self.tick_rate_ns = config.besov_nyquist_interval_ns  # Mínimo tick esperado

        # Precompilar grafos JIT para cada profundidad
        self.signature_graphs: Dict[int, Callable] = {
            2: jax.jit(lambda path: compute_signature(path, depth=2)),
            3: jax.jit(lambda path: compute_signature(path, depth=3)),
            5: jax.jit(lambda path: compute_signature(path, depth=5)),
        }

        # Monitoreo de latencia (EWMA)
        self.latency_history = deque(maxlen=20)  # Últimos 20 ticks
        self.ewma_alpha = 0.3  # Factor de decaimiento
        self.current_depth = 5   # Empezar en máxima resolución
        self.ewma_latency_ns = 0.0

        # Histéresis (evita flapping)
        self.degradation_threshold = 0.7  # Degradar si latencia > 70% del tick-rate
        self.recovery_threshold = 0.5     # Recuperar solo si latencia < 50% del tick-rate

    def update_latency(self, inference_time_ns: int):
        """Actualiza EWMA de latencia y decide si degradar/recuperar."""
        self.latency_history.append(inference_time_ns)

        # EWMA: L_t = alpha * L_actual + (1-alpha) * L_{t-1}
        if self.ewma_latency_ns == 0.0:
            self.ewma_latency_ns = float(inference_time_ns)
        else:
            self.ewma_latency_ns = (
                self.ewma_alpha * inference_time_ns +
                (1 - self.ewma_alpha) * self.ewma_latency_ns
            )

        # Decisión de degradación/recuperación
        latency_ratio = self.ewma_latency_ns / self.tick_rate_ns

        if latency_ratio >= 1.0:  # Crítico: Ya sobrepasamos el tick-rate
```

```
            self.current_depth = 2
        elif latency_ratio >= self.degradation_threshold:  # Advertencia
            self.current_depth = min(self.current_depth, 3)  # Degradar pero no al mínimo
        elif latency_ratio < self.recovery_threshold and self.current_depth < 5:
            # Recuperación gradual (histéresis)
            self.current_depth = min(5, self.current_depth + 1)

    def compute_adaptive_signature(self, path: jnp.ndarray) -> jnp.ndarray:
        """
        Calcula log-signature con profundidad adaptativa.
        """
        # Seleccionar grafo JIT según profundidad actual
        signature_fn = self.signature_graphs[self.current_depth]

        import time
        t_start = time.perf_counter_ns()
        result = signature_fn(path)
        t_end = time.perf_counter_ns()

        # Actualizar estadísticas
        self.update_latency(t_end - t_start)

        return result

    def get_telemetry(self) -> Dict:
        """Devuelve métricas de monitoreo."""
        return {
            "current_depth": self.current_depth,
            "ewma_latency_ms": self.ewma_latency_ns / 1e6,
            "tick_rate_ms": self.tick_rate_ns / 1e6,
            "latency_ratio": self.ewma_latency_ns / self.tick_rate_ns,
            "status": (
                "DEGRADED" if self.current_depth < 5 else "NOMINAL"
            )
        }
```

Análisis de Performance (Profundidad vs Latencia):

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

Trade-off:

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

Integración en UniversalPredictor:

```
class UniversalPredictor:
    def __init__(self, config: PredictorConfig):
        self.config = config
        self.orchestrator = AdaptiveSignatureOrchestrator(config)
        # ...resto de inicialización...

    def predict_next(self, observation: MarketObservation) -> PredictionResult:
        """
        Predicción con Load Shedding automático.
        """
        # 1. Validar entrada
        if not observation.validate_domain():
            return self._fallback_prediction(observation)

        # 2. Construir path incremental
        path = self._build_path(observation)

        # 3. Compute signature con profundidad adaptativa
```

```
        signature = self.orchestrator.compute_adaptive_signature(path)

        # 4. Resto de kernels (A, B, C)
        # ...

        # 5. Retornar con telemetría
        telemetry = self.orchestrator.get_telemetry()
        return PredictionResult(
            predicted_next=predicted_value,
            confidence=confidence_score,
            metadata={
                "signature_depth": telemetry["current_depth"],
                "latency_ms": telemetry["ewma_latency_ms"],
                "system_status": telemetry["status"]
            }
        )
```

Validación en Cisne Negro (Simulación):

```
def test_black_swan_load_shedding():
    """
    Simula evento de cisne negro con tick-rate explosivo.
    """
    config = PredictorConfig(
        log_sig_depth=5,  # Profundidad nominal
        besov_nyquist_interval_ns=100_000_000  # 100ms nominal
    )
    predictor = UniversalPredictor(config)

    # Simular 1000 ticks con aceleración progresiva
    tick_intervals = []
    for i in range(1000):
        if i < 100:  # Primeros 100 ticks: normal (100ms)
            tick_intervals.append(100_000_000)
        elif i < 500:  # Siguientes 400 ticks: aceleración (20ms)
            tick_intervals.append(20_000_000)  # 50 Hz
        else:  # Últimos 500 ticks: crisis (5ms)
            tick_intervals.append(5_000_000)   # 200 Hz

    depths_observed = []
    latencies_observed = []

    for interval_ns in tick_intervals:
        observation = MarketObservation(
            price=jnp.array([random.normal()]),
            target=jnp.array([0.0]),
            timestamp_ns=time.time_ns()
        )

        result = predictor.predict_next(observation)
        depths_observed.append(result.metadata["signature_depth"])
        latencies_observed.append(result.metadata["latency_ms"])

        time.sleep(interval_ns / 1e9)  # Simular tick-rate

    # Assertions
    assert depths_observed[50] == 5, "Debe estar en M=5 durante régimen normal"
    assert depths_observed[300] == 3, "Debe degradar a M=3 durante aceleración"
    assert depths_observed[700] == 2, "Debe degradar a M=2 durante crisis"

    # Verificar que latencia nunca causa backlog infinito
    max_backlog_ratio = max(
        latencies_observed[i] / (tick_intervals[i] / 1e6)
        for i in range(len(tick_intervals))
    )
    assert max_backlog_ratio < 1.2, f"Backlog ratio demasiado alto: {max_backlog_ratio}"

    print(" [] Load Shedding test PASSED")
    print(f"    Profundidad nominal: M=5")
    print(f"    Profundidad mínima observada: M={min(depths_observed)}")
    print(f"    Latencia máxima: {max(latencies_observed):.2f}ms")
    print(f"    Max backlog ratio: {max_backlog_ratio:.2f}x")
```

Beneficios en Producción:

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

Consideraciones Operacionales:

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

Alternativa Avanzada: Poda Selectiva de Kernels
En lugar de solo degradar Rama D, se puede escalar a deshabilitar temporalmente kernels completos:

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

Esto permite **mantener throughput** incluso en crisis extremas (ej. flash crash de mayo 2010).
Conclusión:
El Load Shedding Dinámico es **esencial** para sistemas H.F. que operan en mercados volátiles. Sin esta lógica, eventos de cisne negro producirían backlog infinito y obsolescencia de predicciones. La degradación topológica ($M = 5 \to 3 \to 2$) sacrifica resolución no-lineal profunda pero garantiza que el sistema **nunca se sature**, priorizando latencia sobre precisión extrema cuando es necesario.

16 Telemetría de Jitter (Gatillo Preciso para Load Shedding)

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

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

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

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

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

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

Criterio de Nyquist para Jitter:

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

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

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

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

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

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

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

Solución: Telemetría de Alta Resolución con `perf_counter_ns`

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

```python
import time
import jax.numpy as jnp
from collections import deque
from typing import Deque, Dict

class JitterMonitor:
    """
    Monitor de varianza de latencia (jitter) para detección precisa de saturación.
    Complementa el EWMA de latencia con métrica de dispersión.
    """
    def __init__(self,
                 window_size: int = 100,
                 nyquist_interval_ns: int = 100_000_000,  # 100ms
                 jitter_threshold_ratio: float = 0.8):
        """
        Args:
            window_size: Tamaño de ventana deslizante para calcular jitter
            nyquist_interval_ns: Intervalo mínimo entre ticks (Nyquist limit)
            jitter_threshold_ratio: Umbral de jitter como fracción del Nyquist (ej. 0.8 = 80%)
        """
        self.window_size = window_size
        self.nyquist_interval_ns = nyquist_interval_ns
        self.jitter_threshold_ns = jitter_threshold_ratio * nyquist_interval_ns

        # Historial de latencias (ventana deslizante)
        self.latency_history: Deque[int] = deque(maxlen=window_size)

        # Estadísticas acumuladas
        self.total_predictions = 0
        self.jitter_violations = 0
        self.max_latency_observed_ns = 0

    def record_latency(self, latency_ns: int) -> Dict:
        """
        Registra latencia de predicción y calcula jitter.

        Returns:
            Dict con métricas: jitter_ns, mean_ns, std_ns, violation, etc.
        """
        self.latency_history.append(latency_ns)
        self.total_predictions += 1
        self.max_latency_observed_ns = max(self.max_latency_observed_ns, latency_ns)
```

```
        # Calcular estadísticas solo si hay suficiente historia
        if len(self.latency_history) < 10:
            return {
                "jitter_ns": 0,
                "mean_ns": latency_ns,
                "std_ns": 0,
                "violation": False,
                "degradation_recommended": False
            }

        # Convertir a array numpy para cálculo eficiente
        latencies = jnp.array(list(self.latency_history), dtype=jnp.float64)

        mean_latency = jnp.mean(latencies)
        std_latency = jnp.std(latencies)  # Desviación estándar = jitter

        # Criterio de violación: jitter > 80% del tick-rate
        violation = std_latency > self.jitter_threshold_ns

        if violation:
            self.jitter_violations += 1

        # Recomendación de degradación (combina jitter + media)
        degradation_recommended = (
            violation or
            (mean_latency > 0.7 * self.nyquist_interval_ns and std_latency > 0.5 * self.nyquist_interval_ns)
        )

        return {
            "jitter_ns": float(std_latency),
            "mean_ns": float(mean_latency),
            "std_ns": float(std_latency),
            "violation": bool(violation),
            "degradation_recommended": bool(degradation_recommended),
            "nyquist_limit_ns": self.nyquist_interval_ns,
            "jitter_threshold_ns": self.jitter_threshold_ns,
            "jitter_ratio": float(std_latency / self.nyquist_interval_ns) if self.nyquist_interval_ns > 0 else 0.0
        }

    def get_summary_stats(self) -> Dict:
        """Devuelve resumen estadístico completo."""
        if len(self.latency_history) == 0:
            return {"status": "NO_DATA"}

        latencies = jnp.array(list(self.latency_history), dtype=jnp.float64)

        return {
            "total_predictions": self.total_predictions,
            "jitter_violations": self.jitter_violations,
            "violation_rate": self.jitter_violations / self.total_predictions if self.total_predictions > 0 else 0.0,
            "mean_latency_ms": float(jnp.mean(latencies) / 1e6),
            "std_latency_ms": float(jnp.std(latencies) / 1e6),
            "min_latency_ms": float(jnp.min(latencies) / 1e6),
            "max_latency_ms": float(jnp.max(latencies) / 1e6),
            "p50_latency_ms": float(jnp.percentile(latencies, 50) / 1e6),
            "p95_latency_ms": float(jnp.percentile(latencies, 95) / 1e6),
            "p99_latency_ms": float(jnp.percentile(latencies, 99) / 1e6),
            "max_latency_ever_ms": float(self.max_latency_observed_ns / 1e6)
        }
```

Integración con Load Shedding:

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

```
class AdaptiveSignatureOrchestrator_WithJitter:
    """
    Versión mejorada con telemetría de jitter.
    """
    def __init__(self, config: PredictorConfig):
        self.config = config
```

```
        self.tick_rate_ns = config.besov_nyquist_interval_ns

        # Orquestador original (EWMA de latencia)
        self.signature_graphs = {
            2: jax.jit(lambda path: compute_signature(path, depth=2)),
            3: jax.jit(lambda path: compute_signature(path, depth=3)),
            5: jax.jit(lambda path: compute_signature(path, depth=5)),
        }
        self.current_depth = 5
        self.ewma_latency_ns = 0.0
        self.ewma_alpha = 0.3

        # NUEVO: Monitor de jitter
        self.jitter_monitor = JitterMonitor(
            window_size=100,
            nyquist_interval_ns=self.tick_rate_ns,
            jitter_threshold_ratio=0.8
        )

        # Histéresis mejorada
        self.degradation_threshold = 0.7
        self.recovery_threshold = 0.5

    def compute_adaptive_signature(self, path: jnp.ndarray) -> jnp.ndarray:
        """
        Calcula signature con profundidad adaptativa basada en EWMA + JITTER.
        """
        signature_fn = self.signature_graphs[self.current_depth]

        # Medir latencia con reloj de alta resolución
        t_start = time.perf_counter_ns()
        result = signature_fn(path)
        jax.block_until_ready(result)  # Forzar sincronización GPU
        t_end = time.perf_counter_ns()

        latency_ns = t_end - t_start

        # Actualizar EWMA (como antes)
        if self.ewma_latency_ns == 0.0:
            self.ewma_latency_ns = float(latency_ns)
        else:
            self.ewma_latency_ns = (
                self.ewma_alpha * latency_ns +
                (1 - self.ewma_alpha) * self.ewma_latency_ns
            )

        # NUEVO: Actualizar monitor de jitter
        jitter_metrics = self.jitter_monitor.record_latency(latency_ns)

        # DECISIÓN DE DEGRADACIÓN (criterio combinado: EWMA + Jitter)
        latency_ratio = self.ewma_latency_ns / self.tick_rate_ns
        jitter_violation = jitter_metrics["violation"]
        jitter_ratio = jitter_metrics["jitter_ratio"]

        # Degradación agresiva si jitter es alto
        if jitter_violation and jitter_ratio > 1.0:
            # Jitter crítico: degradar directamente a M=2
            self.current_depth = 2
        elif latency_ratio >= 1.0 or jitter_metrics["degradation_recommended"]:
            # Latencia crítica O jitter recomendado: degradar a M=2
            self.current_depth = 2
        elif latency_ratio >= self.degradation_threshold or jitter_ratio > 0.6:
            # Advertencia: degradar a M=3
            self.current_depth = min(self.current_depth, 3)
        elif latency_ratio < self.recovery_threshold and jitter_ratio < 0.3:
            # Recuperación: ambos indicadores saludables
            self.current_depth = min(5, self.current_depth + 1)

        return result

    def get_telemetry(self) -> Dict:
        """Devuelve métricas combinadas: EWMA + Jitter."""
        jitter_summary = self.jitter_monitor.get_summary_stats()
```

```
        return {
            "current_depth": self.current_depth,
            "ewma_latency_ms": self.ewma_latency_ns / 1e6,
            "tick_rate_ms": self.tick_rate_ns / 1e6,
            "latency_ratio": self.ewma_latency_ns / self.tick_rate_ns,
            "jitter_ms": jitter_summary.get("std_latency_ms", 0.0),
            "jitter_ratio": (jitter_summary.get("std_latency_ms", 0.0) * 1e6) / self.
    tick_rate_ns,
            "jitter_violations": jitter_summary.get("jitter_violations", 0),
            "violation_rate": jitter_summary.get("violation_rate", 0.0),
            "p99_latency_ms": jitter_summary.get("p99_latency_ms", 0.0),
            "status": (
                "DEGRADED_CRITICAL" if self.current_depth == 2 else
                "DEGRADED" if self.current_depth < 5 else
                "NOMINAL"
            )
        }
```

Análisis de Métricas Combinadas:

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

Ventajas del Criterio Dual (EWMA + Jitter):

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

Implementación de Medición Precisa:

```
class UniversalPredictor_WithJitterMonitoring:
    def __init__(self, config: PredictorConfig):
        self.config = config
        self.orchestrator = AdaptiveSignatureOrchestrator_WithJitter(config)
        # ...resto de inicialización...

    def predict_next(self, observation: MarketObservation) -> PredictionResult:
        """
        Predicción con telemetría de jitter de alta resolución.
        """
        # Tiempo de predicción TOTAL (incluye todos los kernels)
        t_prediction_start = time.perf_counter_ns()

        # 1. Validar entrada
        if not observation.validate_domain():
            return self._fallback_prediction(observation)

        # 2. Construir path incremental
        path = self._build_path(observation)

        # 3. Compute signature con profundidad adaptativa (ya mide internamente)
        signature = self.orchestrator.compute_adaptive_signature(path)

        # 4. Resto de kernels (A, B, C)
        # ...

        # 5. Orquestador (fusión + JKO)
```

```
        predicted_value = self.orchestrator.predict_step(observation.price)

        # MEDIR LATENCIA TOTAL
        t_prediction_end = time.perf_counter_ns()
        total_latency_ns = t_prediction_end - t_prediction_start

        # Obtener telemetría
        telemetry = self.orchestrator.get_telemetry()

        # Logging condicional (solo si jitter es alto)
        if telemetry["jitter_ratio"] > 0.6:
            self.logger.warning(
                f"[JITTER ALERT] Ratio={telemetry['jitter_ratio']:.2f}, "
                f"P99={telemetry['p99_latency_ms']:.2f}ms, "
                f"Depth={telemetry['current_depth']}"
            )

        return PredictionResult(
            predicted_next=predicted_value,
            confidence=self._compute_confidence(signature),
            metadata={
                "signature_depth": telemetry["current_depth"],
                "ewma_latency_ms": telemetry["ewma_latency_ms"],
                "jitter_ms": telemetry["jitter_ms"],
                "jitter_ratio": telemetry["jitter_ratio"],
                "p99_latency_ms": telemetry["p99_latency_ms"],
                "total_latency_ns": total_latency_ns,
                "system_status": telemetry["status"]
            }
        )
```

Test de Jitter con Carga Variable:

```
def test_jitter_detection_under_variable_load():
    """
    Simula carga con latencia variable (jitter) para validar detección.
    """
    import random

    config = PredictorConfig(
        log_sig_depth=5,
        besov_nyquist_interval_ns=100_000_000  # 100ms
    )
    predictor = UniversalPredictor_WithJitterMonitoring(config)

    # Simular 200 predicciones con patrones de latencia diferentes
    latencies_observed = []
    depths_observed = []

    for i in range(200):
        # FASE 1 (i < 50): Latencia estable (0.8-1.2ms)
        if i < 50:
            simulated_processing_time_ns = random.uniform(800_000, 1_200_000)

        # FASE 2 (50 <= i < 100): Jitter alto (0.5-30ms)
        elif i < 100:
            simulated_processing_time_ns = random.uniform(500_000, 30_000_000)

        # FASE 3 (100 <= i < 150): Latencia alta constante (15-20ms)
        elif i < 150:
            simulated_processing_time_ns = random.uniform(15_000_000, 20_000_000)

        # FASE 4 (150 <= i < 200): Recuperación gradual (1-3ms)
        else:
            simulated_processing_time_ns = random.uniform(1_000_000, 3_000_000)

        # Simular predicción
        observation = MarketObservation(
            price=jnp.array([random.normal()]),
            target=jnp.array([0.0]),
            timestamp_ns=time.time_ns()
        )

        # Inyectar latencia artificial
```

```
        time.sleep(simulated_processing_time_ns / 1e9)

        result = predictor.predict_next(observation)

        latencies_observed.append(result.metadata["total_latency_ns"] / 1e6)
        depths_observed.append(result.metadata["signature_depth"])

    # Assertions
    # FASE 1: Sistema debe mantener M=5 (jitter bajo)
    assert all(d == 5 for d in depths_observed[10:50]), "Debería mantener M=5 en fase estable"

    # FASE 2: Sistema debe degradar por jitter alto
    assert any(d < 5 for d in depths_observed[60:100]), "Debería degradar por jitter alto"

    # FASE 3: Sistema debe degradar por latencia alta
    assert all(d <= 3 for d in depths_observed[120:150]), "Debería degradar a M<=3 por
    latencia alta"

    # FASE 4: Sistema debe recuperar gradualmente
    final_depth = depths_observed[-10:]
    assert any(d >= 4 for d in final_depth), "Debería recuperar hacia M=5"

    # Estadísticas finales
    telemetry = predictor.orchestrator.get_telemetry()

    print(" [] Jitter detection test PASSED")
    print(f"    Jitter violations: {telemetry['jitter_violations']}")
    print(f"    Violation rate: {telemetry['violation_rate']:.2%}")
    print(f"    P99 latency: {telemetry['p99_latency_ms']:.2f}ms")
    print(f"    Final depth: {telemetry['current_depth']}")
    print(f"    Final jitter ratio: {telemetry['jitter_ratio']:.2f}")
```

Dashboard de Monitoreo (Prometheus + Grafana):

```
from prometheus_client import Histogram, Gauge, Counter

class UniversalPredictor_WithMetrics:
    def __init__(self, config: PredictorConfig):
        # ...inicialización...

        # Métricas de latencia
        self.latency_histogram = Histogram(
            'predictor_latency_seconds',
            'Latency distribution of predictions',
            buckets=[0.001, 0.005, 0.010, 0.050, 0.100, 0.500, 1.0]
        )

        # Métricas de jitter
        self.jitter_gauge = Gauge(
            'predictor_jitter_milliseconds',
            'Standard deviation of latency (jitter)'
        )
        self.jitter_ratio_gauge = Gauge(
            'predictor_jitter_ratio',
            'Jitter as ratio of Nyquist limit'
        )
        self.jitter_violations_counter = Counter(
            'predictor_jitter_violations_total',
            'Total number of jitter threshold violations'
        )

        # Métricas de profundidad
        self.signature_depth_gauge = Gauge(
            'predictor_signature_depth',
            'Current signature computation depth (M)'
        )

    def predict_next(self, observation: MarketObservation) -> PredictionResult:
        with self.latency_histogram.time():
            result = self._predict_internal(observation)

        # Actualizar métricas de jitter
        telemetry = self.orchestrator.get_telemetry()
        self.jitter_gauge.set(telemetry["jitter_ms"])
```

```
        self.jitter_ratio_gauge.set(telemetry["jitter_ratio"])
        self.signature_depth_gauge.set(telemetry["current_depth"])

        if telemetry["jitter_ratio"] > 0.8:
            self.jitter_violations_counter.inc()

        return result
```

Alertas Operacionales (Recomendadas):

```
# prometheus_alerts.yml
groups:
- name: predictor_latency
  interval: 30s
  rules:
  - alert: PredictorHighJitter
    expr: predictor_jitter_ratio > 0.8
    for: 1m
    labels:
      severity: warning
    annotations:
      summary: "High jitter detected in predictor"
      description: "Jitter ratio {{ $value }} exceeds 80% of Nyquist limit"

  - alert: PredictorJitterCritical
    expr: predictor_jitter_ratio > 1.2
    for: 30s
    labels:
      severity: critical
    annotations:
      summary: "Critical jitter in predictor"
      description: "Jitter ratio {{ $value }} exceeds Nyquist limit, predictions may be stale"

  - alert: PredictorDegraded
    expr: predictor_signature_depth < 5
    for: 5m
    labels:
      severity: info
    annotations:
      summary: "Predictor operating in degraded mode"
      description: "Signature depth reduced to {{ $value }} due to load"
```

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

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

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

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

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

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