Python API Specification - Universal Predictor

Software Engineering

February 19, 2026

Contents

1 Introduction

This document specifies the Python implementation of the abstract I/O interface defined in *Stochastic_Predictor_IO*. The API exposes the `UniversalPredictor` class for high-performance environments using JAX for numerical acceleration.

2 Data Structures (Typing)

We use `dataclasses` and `jaxtyping` to enforce immutability and strict dimensional typing for tensors.

2.1 Configuration (Λ)

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

@dataclass(frozen=True)
class PredictorConfig:
    """Hyperparameter vector Lambda."""
    schema_version: str = "1.0"   # Snapshot versioning
    epsilon: float = 1e-3         # Entropic regularization (Sinkhorn)
    learning_rate: float = 0.01   # JKO learning rate
    log_sig_depth: int = 3        # Signature depth (Kernel D)
    wtmm_buffer_size: int = 128   # WTMM buffer size
    besov_cone_c: float = 1.5     # Besov cone influence
    holder_threshold: float = 0.4 # Circuit breaker threshold
    cusum_h: float = 5.0          # CUSUM threshold
    cusum_k: float = 0.5          # CUSUM slack
    grace_period_steps: int = 20  # Post-regime refractory period
    volatility_alpha: float = 0.1 # EMA decay for variance

    # Load shedding and anti-aliasing
    staleness_ttl_ns: int = 500_000_000          # TTL (500ms)
    besov_nyquist_interval_ns: int = 100_000_000 # Nyquist soft limit (100ms)
    inference_recovery_hysteresis: float = 0.8  # Degraded mode recovery factor
```

2.2 Operational Input $(y_t, y_{reference}, \tau)$

```python
@dataclass(frozen=True)
class ProcessState:
    magnitude: Float[Array, "1"]  # y_t (normalized or absolute)
    reference: Float[Array, "1"]  # y_reference
    timestamp_ns: int             # Unix epoch (nanoseconds)

    def validate_domain(self, sigma_bound: float = 20.0, sigma_val: float = 1.0) -> bool:
        """Catastrophic outlier detection (> 20 sigma)."""
        return abs(self.magnitude) <= (sigma_bound * sigma_val)
```

2.3 System Output

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

    # Telemetry
    holder_exponent: Float[Array, "1"]
    cusum_drift: Float[Array, "1"]
    distance_to_collapse: Float[Array, "1"]
    free_energy: Float[Array, "1"]

    # Advanced telemetry
    kurtosis: Float[Array, "1"]
    dgm_entropy: Float[Array, "1"]
    adaptive_threshold: Float[Array, "1"]

    # Orchestrator state
```

```python
    weights: Float[Array, "4"]

    # Health flags
    sinkhorn_converged: Bool[Array, "1"]
    degraded_inference_mode: bool
    emergency_mode: bool
    regime_change_detected: bool
    mode_collapse_warning: bool

    mode: str  # "Standard" | "Robust" | "Emergency"
```

3 Multi-Tenant Architecture (Stateless Functional Pattern)

To support hundreds of assets on a single server, the API exposes a purely functional mode. This allows state management in low-latency external storage (Redis) while sharing the compiled JAX graph across assets.

3.1 Throughput Maximization (Vectorized Batching)

This architecture enables `jax.vmap` to batch multiple asset states in a single hardware call, minimizing the Python GIL impact and maximizing GPU occupancy.

```python
class FunctionalPredictor:
    """
    Stateless implementation for JAX core.
    Scales to thousands of predictors sharing the same graph.
    """
    def __init__(self, config: PredictorConfig):
        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):
        """Create a zeroed cold-state structure."""
        return self._initialize_state_structure()

    def step(self, state, obs: ProcessState) -> tuple[object, PredictionResult]:
        """
        Pure state transition: (S_t, Obs_t) -> (S_{t+1}, Pred_{t+1})
        """
        should_freeze = self._should_freeze(obs)
        new_state, raw_result = self._jit_update(
            state,
            obs.magnitude,
            obs.reference,
            freeze_weights=should_freeze
        )
        result = PredictionResult(
            predicted_next=raw_result.y_next,
            # map the remaining fields
        )
        return new_state, result

    def step_batch(self, states, obs_batch: ProcessState):
        """Vectorized batch processing for N assets."""
        freeze_flags = self._should_freeze_batch(obs_batch)
        new_states, results = self._vmap_update(
            states, obs_batch.magnitude, obs_batch.reference, freeze_flags
        )
        return new_states, results
```

4 Main Class: `UniversalPredictor` (Stateful Wrapper)

This class wraps the functional pattern for single-tenant usage with state held in local memory.

4.1 Initialization

```python
class UniversalPredictor:
    def __init__(self, config: PredictorConfig):
        """
        Initialize the JAX compute graph (XLA JIT compilation).
        Allocate static device buffers (VRAM).
        Internal state stores persistent rolling buffers updated with
        functional ops to avoid CPU<->VRAM transfers.
        """
        self.config = config
        self._state = self._initialize_state()
        self._jit_update = jax.jit(self._core_update_step)
        self._last_timestamp_ns = 0

    def fit_history(self, history: list[float]) -> bool:
        """
        Cold-start bootstrapping. Requires at least N_buf samples.
        Returns True if Sinkhorn and CUSUM converge.
        """
        if len(history) < self.config.wtmm_buffer_size:
            raise ValueError(f"Insufficient history. Required: {self.config.wtmm_buffer_size}")

        self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))

        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 finished without stable convergence.")
            return False
        return True
```

4.2 Execution Method ($t \to t + 1$)

```python
    def step(self, obs: ProcessState) -> PredictionResult:
        """Execute one prediction cycle with domain and TTL validation."""
        if not obs.validate_domain():
            logger.error("Catastrophic outlier detected. Ignoring tick.")
            return self._last_valid_result

        current_time = time.time_ns()
        latency = current_time - obs.timestamp_ns
        is_stale = latency > self.config.staleness_ttl_ns

        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: interval {dt_arrival}ns > Nyquist limit. WTMM may alias."
            )

        self._last_timestamp_ns = obs.timestamp_ns
        should_freeze = is_stale or is_sparse

        new_state, result_data = self._jit_update(
            self._state,
            obs.magnitude,
            obs.reference,
            freeze_weights=should_freeze,
        )
        self._state = new_state

        return PredictionResult(
            predicted_next=result_data.y_next,
            holder_exponent=result_data.H_t,
            sinkhorn_converged=result_data.converged,
            # map remaining fields
        )
```

5 Preventing VRAM Fragmentation (JAX Memory Management)

Production problem: JAX preallocates 90% of GPU memory on first access. Long-running systems may fragment VRAM and hit silent OOM after weeks.

Solution: Configure environment variables **before** importing JAX:

```python
import os

os.environ['XLA_PYTHON_CLIENT_MEM_FRACTION'] = '0.7'
os.environ['XLA_PYTHON_CLIENT_ALLOCATOR'] = 'platform'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'

import jax
import jax.numpy as jnp
```

6 VRAM Monitoring

```python
import psutil
import subprocess

def monitor_vram_fragmentation(interval_seconds=60):
    """Background thread for VRAM monitoring."""
    import time
    import threading

    def _monitor():
        while True:
            try:
                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()
```

7 Recommended Deployment Configuration

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

export XLA_PYTHON_CLIENT_MEM_FRACTION=0.7
export XLA_PYTHON_CLIENT_ALLOCATOR=platform
export TF_FORCE_GPU_ALLOW_GROWTH=true

echo "[INFO] XLA VRAM Fraction: 0.7 (28/40 GB on A100)"
echo "[INFO] Allocator: platform (dynamic)"
echo "[INFO] GPU growth: enabled"

python3 -u predictor_service.py \
    --config config.yaml \
    --device gpu \
    --pool-size 100 \
    --monitor-interval 300
```

8 Persistence (Atomic Snapshotting)

```python
import hashlib
import msgpack

def save_snapshot(self, filepath: str):
    """
    Export internal state Sigma_t as MessagePack.
    Append SHA-256 checksum at the end of the file.
    """
    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)
    checksum = hashlib.sha256(data_bytes).hexdigest()

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


def load_snapshot(self, filepath: str):
    """
    Load state. Validate SHA-256 and schema_version.
    Raise ValueError if validation fails.
    """
    with open(filepath, "rb") as f:
        content = f.read()

    data_bytes = content[:-64]
    stored_checksum = content[-64:].decode('utf-8')

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

    payload = msgpack.unpackb(data_bytes)
    loaded_schema = payload.get('schema_version', 'unknown')
    if loaded_schema != self.config.schema_version:
        raise ValueError(
            f"Schema version mismatch: snapshot={loaded_schema}, current={self.config.schema_version}."
        )

    self._state = self._deserialize_jax_state(payload)
```

9 Asynchronous I/O for Snapshots (Non-Blocking)

```python
import concurrent.futures
```

```python
import hashlib
import msgpack
import threading
import time

class UniversalPredictor_AsyncIO:
    def __init__(self, n_worker_threads=2):
        self.io_executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=n_worker_threads,
            thread_name_prefix="snapshot_io_"
        )
        self.pending_snapshot_future = None
        self.snapshot_lock = threading.Lock()

    def _compute_and_save_async(self, filepath: str, data_bytes: bytes):
        checksum = hashlib.sha256(data_bytes).hexdigest()
        temp_filepath = filepath + ".tmp"
        try:
            with open(temp_filepath, "wb") as f:
                f.write(data_bytes)
                f.write(checksum.encode('utf-8'))
            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:
        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)
        future = self.io_executor.submit(self._compute_and_save_async, filepath, data_bytes)
        with self.snapshot_lock:
            self.pending_snapshot_future = future
        return future
```

10 Graceful Shutdown for Containers

```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)
        self.shutdown_requested = threading.Event()
        self.is_accepting_data = True
        self.input_buffer_lock = threading.Lock()
        self.residual_buffer = []

        signal.signal(signal.SIGTERM, self._handle_sigterm)
        signal.signal(signal.SIGINT, self._handle_sigint)

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

    def _handle_sigterm(self, signum, frame):
        self.logger.warning(f"[SIGTERM] Received signal {signum}. Initiating graceful shutdown
    ...")
        self.shutdown_requested.set()

    def _handle_sigint(self, signum, frame):
        self.logger.warning(f"[SIGINT] Received signal {signum}. Initiating graceful shutdown
    ...")
        self.shutdown_requested.set()

    def accept_observation(self, obs: ProcessState) -> Optional[PredictionResult]:
        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)
        return self._process_observation(obs)

    def _process_observation(self, obs: ProcessState) -> PredictionResult:
        result = self.predictor.predict_next(obs)
        with self.input_buffer_lock:
            if obs in self.residual_buffer:
                self.residual_buffer.remove(obs)
        return result

    def graceful_shutdown(self, timeout_seconds: int = 25):
        shutdown_start = time.time()
        self.logger.info("GRACEFUL SHUTDOWN INITIATED")
        self.is_accepting_data = False
        time.sleep(0.1)

        with self.input_buffer_lock:
            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}")

        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)
            except Exception as e:
                self.logger.error(f"Async snapshot failed: {e}")

        try:
```

```python
            final_snapshot_path = f"snapshots/shutdown_{int(time.time())}.pkl"
            self.predictor.save_snapshot(final_snapshot_path)
        except Exception as e:
            self.logger.error(f"Final snapshot failed: {e}")

        try:
            if hasattr(self.predictor, 'io_executor'):
                self.predictor.io_executor.shutdown(wait=True, cancel_futures=False)
        except Exception as e:
            self.logger.error(f"Error closing resources: {e}")

        total_time = time.time() - shutdown_start
        self.logger.info(f"SHUTDOWN COMPLETED ({total_time:.2f}s)")
        sys.exit(0)
```

11 Prometheus Integration

```python
from prometheus_client import Counter, Histogram

class UniversalPredictor_GracefulShutdown_Monitored:
    def __init__(self, config: PredictorConfig):
        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]
        )
```

12 Adaptive CUSUM Threshold

The system implements the adaptive threshold lemma based on kurtosis:

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

13 Grace Period (Post-Regime Refractory Window)

After a regime change ($G^+ > h_t$), the system resets weights to uniform. A grace period prevents a cascade of false alarms while weights re-converge. The detector continues to compute G^+ but does not emit an alarm until the counter expires.

14 Operational Flags and Recovery

The system exposes explicit flags:

- **degraded_inference_mode**: TTL exceeded; weights frozen.
- **emergency_mode**: $H_t < H_{min}$; force Kernel D and Huber loss.
- **regime_change_detected**: CUSUM alarm; entropy reset.
- **mode_collapse_warning**: DGM entropy below threshold for > 10 steps.

15 Error Handling and Exceptions

Standard alerts:

- `DomainError`: catastrophic outlier $> 20\sigma$
- `StalenessWarning`: TTL exceeded
- `FrequencyWarning`: Nyquist limit violated
- `IntegrityError`: snapshot verification failed

16 Production Logging Example

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

def save_emergency_dump(predictor, result, asset_id: str):
    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 forensic analysis")
```

17 Deterministic Floating-Point Reproducibility

Configure deterministic reductions and PRNG before importing JAX:

```python
import os
import numpy as np
import jax

os.environ['XLA_FLAGS'] = '--xla_cpu_use_cross_replica_callbacks=false'
os.environ['JAX_DETERMINISTIC_REDUCTIONS'] = '1'
os.environ['JAX_TRACEBACK_FILTERING'] = 'off'

np.random.seed(42)

jax.config.update('jax_default_prng_impl', 'threefry2x32')
key = jax.random.PRNGKey(42)

jax.config.update('jax_enable_x64', True)
```

18 Load Shedding (Adaptive Topological Pruning)

When tick rate spikes, dynamically reduce signature depth M based on EWMA latency and jitter. Precompile multiple JIT graphs for $M \in \{2, 3, 5\}$ and switch by thresholds to prevent backlog.

19 Jitter Telemetry

Measure latency jitter using `time.perf_counter_ns()` and degrade if jitter exceeds 80% of Nyquist limit. Expose P95/P99 in telemetry and Prometheus.

20 Dependency Pinning

Strict version pinning is mandatory. Any update must be tested for bit-exact parity and documented. Use exact versions in `requirements.txt` and `environment.yml`, never open ranges.