

Python API Specification - Universal Predictor

Software Engineering

February 19, 2026

Contents

1	Introduction	3
2	Data Structures (Typing)	3
2.1	Configuration (Λ)	3
2.2	Operational Input ($y_t, y_{reference}, \tau$)	3
2.3	System Output	3
3	Multi-Tenant Architecture (Stateless Functional Pattern)	4
3.1	Throughput Maximization (Vectorized Batching)	4
4	Main Class: UniversalPredictor (Stateful Wrapper)	4
4.1	Initialization	5
4.2	Execution Method ($t \rightarrow t + 1$)	5
5	Preventing VRAM Fragmentation (JAX Memory Management)	6
6	VRAM Monitoring	6
7	Recommended Deployment Configuration	6
8	Persistence (Atomic Snapshotting)	7
9	Asynchronous I/O for Snapshots (Non-Blocking)	7
10	Graceful Shutdown for Containers	8
11	Prometheus Integration	10
12	Adaptive CUSUM Threshold	10
13	Grace Period (Post-Regime Refractory Window)	10
14	Operational Flags and Recovery	10
15	Error Handling and Exceptions	11
16	Production Logging Example	11
17	Deterministic Floating-Point Reproducibility	11
18	Load Shedding (Adaptive Topological Pruning)	12
19	Jitter Telemetry	12
20	Dependency Pinning	12

21 Meta-Optimization API (Bayesian Hyperparameter Tuning)	12
21.1 BayesianMetaOptimizer Class	12
21.2 OptimizationResult Schema (Pydantic)	16
21.3 Non-Blocking I/O Execution Pattern	18
21.4 Integration with Prediction Pipeline	20
22 Dependency Pinning	21

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)

```
1 from dataclasses import dataclass
2 from typing import Optional
3 from jaxtyping import Float, Array, Bool
4
5 @dataclass(frozen=True)
6 class PredictorConfig:
7     """Hyperparameter vector Lambda."""
8     schema_version: str = "1.0" # Snapshot versioning
9     epsilon: float = 1e-3 # Entropic regularization (Sinkhorn)
10    learning_rate: float = 0.01 # JKO learning rate
11    log_sig_depth: int = 3 # Signature depth (Kernel D)
12    wtmm_buffer_size: int = 128 # WTMM buffer size
13    besov_cone_c: float = 1.5 # Besov cone influence
14    holder_threshold: float = 0.4 # Circuit breaker threshold
15    cusum_h: float = 5.0 # CUSUM threshold
16    cusum_k: float = 0.5 # CUSUM slack
17    grace_period_steps: int = 20 # Post-regime refractory period
18    volatility_alpha: float = 0.1 # EMA decay for variance
19
20    # Load shedding and anti-aliasing
21    staleness_ttl_ns: int = 500_000_000 # TTL (500ms)
22    besov_nyquist_interval_ns: int = 100_000_000 # Nyquist soft limit (100ms)
23    inference_recovery_hysteresis: float = 0.8 # Degraded mode recovery factor
```

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

```
1 @dataclass(frozen=True)
2 class ProcessState:
3     magnitude: Float[Array, "1"] # y_t (normalized or absolute)
4     reference: Float[Array, "1"] # y_reference
5     timestamp_ns: int # Unix epoch (nanoseconds)
6
7     def validate_domain(self, sigma_bound: float = 20.0, sigma_val: float = 1.0) -> bool:
8         """Catastrophic outlier detection (> 20 sigma)."""
9         return abs(self.magnitude) <= (sigma_bound * sigma_val)
```

2.3 System Output

```
1 @dataclass(frozen=True)
2 class PredictionResult:
3     predicted_next: Float[Array, "1"] # y_{t+1} (Z-score)
4
5     # Telemetry
6     holder_exponent: Float[Array, "1"]
7     cusum_drift: Float[Array, "1"]
8     distance_to_collapse: Float[Array, "1"]
9     free_energy: Float[Array, "1"]
10
11    # Advanced telemetry
12    kurtosis: Float[Array, "1"]
13    dgm_entropy: Float[Array, "1"]
14    adaptive_threshold: Float[Array, "1"]
15
16    # Orchestrator state
```

```

17 weights: Float[Array, "4"]
18
19 # Health flags
20 sinkhorn_converged: Bool[Array, "1"]
21 degraded_inference_mode: bool
22 emergency_mode: bool
23 regime_change_detected: bool
24 mode_collapse_warning: bool
25
26 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.

```

1 class FunctionalPredictor:
2     """
3     Stateless implementation for JAX core.
4     Scales to thousands of predictors sharing the same graph.
5     """
6     def __init__(self, config: PredictorConfig):
7         self.config = config
8         self._core_step = self._core_update_step
9         self._jit_update = jax.jit(self._core_step)
10        self._vmap_update = jax.jit(jax.vmap(self._core_step, in_axes=(0, 0, 0, 0)))
11
12    def init_state(self):
13        """Create a zeroed cold-state structure."""
14        return self._initialize_state_structure()
15
16    def step(self, state, obs: ProcessState) -> tuple[object, PredictionResult]:
17        """
18        Pure state transition: (St, Obst) -> (S{t+1}, Pred{t+1})
19        """
20        should_freeze = self._should_freeze(obs)
21        new_state, raw_result = self._jit_update(
22            state,
23            obs.magnitude,
24            obs.reference,
25            freeze_weights=should_freeze
26        )
27        result = PredictionResult(
28            predicted_next=raw_result.y_next,
29            # map the remaining fields
30        )
31        return new_state, result
32
33    def step_batch(self, states, obs_batch: ProcessState):
34        """Vectorized batch processing for N assets."""
35        freeze_flags = self._should_freeze_batch(obs_batch)
36        new_states, results = self._vmap_update(
37            states, obs_batch.magnitude, obs_batch.reference, freeze_flags
38        )
39        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

```
1 class UniversalPredictor:
2     def __init__(self, config: PredictorConfig):
3         """
4         Initialize the JAX compute graph (XLA JIT compilation).
5         Allocate static device buffers (VRAM).
6         Internal state stores persistent rolling buffers updated with
7         functional ops to avoid CPU<->VRAM transfers.
8         """
9         self.config = config
10        self._state = self._initialize_state()
11        self._jit_update = jax.jit(self._core_update_step)
12        self._last_timestamp_ns = 0
13
14    def fit_history(self, history: list[float]) -> bool:
15        """
16        Cold-start bootstrapping. Requires at least N_buf samples.
17        Returns True if Sinkhorn and CUSUM converge.
18        """
19        if len(history) < self.config.wtmm_buffer_size:
20            raise ValueError(f"Insufficient history. Required: {self.config.wtmm_buffer_size}")
21
22        self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))
23
24        is_converged = final_metrics.sinkhorn_converged
25        is_stable = final_metrics.cusum_drift < self.config.cusum_h
26        if not (is_converged and is_stable):
27            logger.warning("Cold start finished without stable convergence.")
28            return False
29        return True
```

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

```
1 def step(self, obs: ProcessState) -> PredictionResult:
2     """Execute one prediction cycle with domain and TTL validation."""
3     if not obs.validate_domain():
4         logger.error("Catastrophic outlier detected. Ignoring tick.")
5         return self._last_valid_result
6
7     current_time = time.time_ns()
8     latency = current_time - obs.timestamp_ns
9     is_stale = latency > self.config.staleness_ttl_ns
10
11    dt_arrival = obs.timestamp_ns - self._last_timestamp_ns
12    is_sparse = (self._last_timestamp_ns > 0) and (
13        dt_arrival > self.config.besov_nyquist_interval_ns
14    )
15    if is_sparse:
16        logger.warning(
17            f"FrequencyWarning: interval {dt_arrival}ns > Nyquist limit. WTMM may alias."
18        )
19
20    self._last_timestamp_ns = obs.timestamp_ns
21    should_freeze = is_stale or is_sparse
22
23    new_state, result_data = self._jit_update(
24        self._state,
25        obs.magnitude,
26        obs.reference,
27        freeze_weights=should_freeze,
28    )
29    self._state = new_state
30
31    return PredictionResult(
32        predicted_next=result_data.y_next,
33        holder_exponent=result_data.H_t,
34        sinkhorn_converged=result_data.converged,
35        # map remaining fields
36    )
```

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:

```
1 import os
2
3 os.environ['XLA_PYTHON_CLIENT_MEM_FRACTION'] = '0.7'
4 os.environ['XLA_PYTHON_CLIENT_ALLOCATOR'] = 'platform'
5 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
6
7 import jax
8 import jax.numpy as jnp
```

6 VRAM Monitoring

```
1 import psutil
2 import subprocess
3
4 def monitor_vram_fragmentation(interval_seconds=60):
5     """Background thread for VRAM monitoring."""
6     import time
7     import threading
8
9     def _monitor():
10         while True:
11             try:
12                 result = subprocess.run(
13                     ['nvidia-smi', '--query-gpu=memory.used,memory.total',
14                     '--format=csv,nounits,noheader'],
15                     capture_output=True, text=True, timeout=5
16                 )
17                 if result.returncode == 0:
18                     used, total = map(float, result.stdout.strip().split(','))
19                     utilization = 100.0 * used / total
20                     if utilization > 0.95:
21                         print(f"[WARNING] VRAM near saturation: {utilization:.1f}%")
22                     elif utilization > 0.85:
23                         print(f"[INFO] VRAM utilization: {utilization:.1f}% (elevated)")
24                     time.sleep(interval_seconds)
25             except Exception as e:
26                 print(f"[ERROR] VRAM monitoring failed: {e}")
27                 break
28
29     thread = threading.Thread(target=_monitor, daemon=True)
30     thread.start()
```

7 Recommended Deployment Configuration

```
1 #!/bin/bash
2 # deployment/run_predictor.sh
3
4 export XLA_PYTHON_CLIENT_MEM_FRACTION=0.7
5 export XLA_PYTHON_CLIENT_ALLOCATOR=platform
6 export TF_FORCE_GPU_ALLOW_GROWTH=true
7
8 echo "[INFO] XLA VRAM Fraction: 0.7 (28/40 GB on A100)"
9 echo "[INFO] Allocator: platform (dynamic)"
10 echo "[INFO] GPU growth: enabled"
11
12 python3 -u predictor_service.py \
13     --config config.yaml \
14     --device gpu \
15     --pool-size 100 \
16     --monitor-interval 300
```

8 Persistence (Atomic Snapshotting)

```
1 import hashlib
2 import msgpack
3
4 def save_snapshot(self, filepath: str):
5     """
6     Export internal state Sigma_t as MessagePack.
7     Append SHA-256 checksum at the end of the file.
8     """
9     state_dict = self._serialize_jax_state(self._state)
10    payload = {
11        "schema_version": self.config.schema_version,
12        "timestamp": time.time_ns(),
13        "config": asdict(self.config),
14        "global": state_dict["global"],
15        "telemetry": {
16            "kurtosis": float(self._state.kurtosis),
17            "dgm_entropy": float(self._state.dgm_entropy),
18            "adaptive_threshold": float(self._state.h_adaptive)
19        },
20        "flags": {
21            "degraded_inference": bool(self._state.degraded_mode),
22            "emergency": bool(self._state.emergency_mode),
23            "regime_change": bool(self._state.regime_changed),
24            "mode_collapse": bool(self._state.mode_collapse_warning)
25        },
26        "kernels": {
27            "A": state_dict["kernel_a"],
28            "B": state_dict["kernel_b"],
29            "C": state_dict["kernel_c"],
30            "D": state_dict["kernel_d"]
31        }
32    }
33    data_bytes = msgpack.packb(payload)
34    checksum = hashlib.sha256(data_bytes).hexdigest()
35
36    with open(filepath, "wb") as f:
37        f.write(data_bytes)
38        f.write(checksum.encode('utf-8'))
39
40
41 def load_snapshot(self, filepath: str):
42     """
43     Load state. Validate SHA-256 and schema_version.
44     Raise ValueError if validation fails.
45     """
46     with open(filepath, "rb") as f:
47         content = f.read()
48
49     data_bytes = content[:-64]
50     stored_checksum = content[-64:].decode('utf-8')
51
52     computed = hashlib.sha256(data_bytes).hexdigest()
53     if computed != stored_checksum:
54         raise ValueError("Snapshot corrupt: checksum mismatch.")
55
56     payload = msgpack.unpackb(data_bytes)
57     loaded_schema = payload.get('schema_version', 'unknown')
58     if loaded_schema != self.config.schema_version:
59         raise ValueError(
60             f"Schema version mismatch: snapshot={loaded_schema}, current={self.config.schema_version}."
61         )
62
63     self._state = self._deserialize_jax_state(payload)
```

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

```
1 import concurrent.futures
```

```

2 import hashlib
3 import msgpack
4 import threading
5 import time
6
7 class UniversalPredictor_AsyncIO:
8     def __init__(self, n_worker_threads=2):
9         self.io_executor = concurrent.futures.ThreadPoolExecutor(
10             max_workers=n_worker_threads,
11             thread_name_prefix="snapshot_io_"
12         )
13         self.pending_snapshot_future = None
14         self.snapshot_lock = threading.Lock()
15
16     def _compute_and_save_async(self, filepath: str, data_bytes: bytes):
17         checksum = hashlib.sha256(data_bytes).hexdigest()
18         temp_filepath = filepath + ".tmp"
19         try:
20             with open(temp_filepath, "wb") as f:
21                 f.write(data_bytes)
22                 f.write(checksum.encode('utf-8'))
23             import os
24             os.replace(temp_filepath, filepath)
25             return {
26                 'status': 'success',
27                 'filepath': filepath,
28                 'filesize_bytes': len(data_bytes),
29                 'checksum': checksum,
30                 'timestamp': time.time()
31             }
32         except Exception as e:
33             return {
34                 'status': 'error',
35                 'filepath': filepath,
36                 'error': str(e),
37                 'timestamp': time.time()
38             }
39
40     def save_snapshot_nonblocking(self, filepath: str) -> concurrent.futures.Future:
41         state_dict = self._serialize_jax_state(self._state)
42         payload = {
43             "schema_version": self.config.schema_version,
44             "timestamp": time.time_ns(),
45             "config": asdict(self.config),
46             "global": state_dict["global"],
47             "telemetry": {
48                 "kurtosis": float(self._state.kurtosis),
49                 "dgm_entropy": float(self._state.dgm_entropy),
50                 "adaptive_threshold": float(self._state.h_adaptive)
51             },
52             "flags": {
53                 "degraded_inference": bool(self._state.degraded_mode),
54                 "emergency": bool(self._state.emergency_mode),
55                 "regime_change": bool(self._state.regime_changed),
56                 "mode_collapse": bool(self._state.mode_collapse_warning)
57             },
58             "kernels": {
59                 "A": state_dict["kernel_a"],
60                 "B": state_dict["kernel_b"],
61                 "C": state_dict["kernel_c"],
62                 "D": state_dict["kernel_d"]
63             }
64         }
65         data_bytes = msgpack.packb(payload)
66         future = self.io_executor.submit(self._compute_and_save_async, filepath, data_bytes)
67         with self.snapshot_lock:
68             self.pending_snapshot_future = future
69         return future

```

10 Graceful Shutdown for Containers


```

1 import signal
2 import sys
3 import threading
4 import time
5 import logging
6 from typing import Optional
7
8 class UniversalPredictor_GracefulShutdown:
9     def __init__(self, config: PredictorConfig):
10         self.config = config
11         self.predictor = UniversalPredictor_AsyncIO(config)
12         self.shutdown_requested = threading.Event()
13         self.is_accepting_data = True
14         self.input_buffer_lock = threading.Lock()
15         self.residual_buffer = []
16
17         signal.signal(signal.SIGTERM, self._handle_sigterm)
18         signal.signal(signal.SIGINT, self._handle_sigint)
19
20         self.logger = logging.getLogger("predictor.shutdown")
21         self.logger.info("[INIT] Graceful shutdown handler registered")
22
23     def _handle_sigterm(self, signum, frame):
24         self.logger.warning(f"[SIGTERM] Received signal {signum}. Initiating graceful shutdown
25         ...")
26         self.shutdown_requested.set()
27
28     def _handle_sigint(self, signum, frame):
29         self.logger.warning(f"[SIGINT] Received signal {signum}. Initiating graceful shutdown
30         ...")
31         self.shutdown_requested.set()
32
33     def accept_observation(self, obs: ProcessState) -> Optional[PredictionResult]:
34         if self.shutdown_requested.is_set() or not self.is_accepting_data:
35             self.logger.warning(f"[REJECT] Observation rejected (shutdown in progress): {obs.
36             timestamp_ns}")
37             return None
38         with self.input_buffer_lock:
39             self.residual_buffer.append(obs)
40         return self._process_observation(obs)
41
42     def _process_observation(self, obs: ProcessState) -> PredictionResult:
43         result = self.predictor.predict_next(obs)
44         with self.input_buffer_lock:
45             if obs in self.residual_buffer:
46                 self.residual_buffer.remove(obs)
47         return result
48
49     def graceful_shutdown(self, timeout_seconds: int = 25):
50         shutdown_start = time.time()
51         self.logger.info("GRACEFUL SHUTDOWN INITIATED")
52         self.is_accepting_data = False
53         time.sleep(0.1)
54
55         with self.input_buffer_lock:
56             for obs in list(self.residual_buffer):
57                 if time.time() - shutdown_start > timeout_seconds - 10:
58                     self.logger.warning("Timeout approaching, aborting residual processing")
59                     break
60                 try:
61                     _ = self._process_observation(obs)
62                 except Exception as e:
63                     self.logger.error(f"Error processing residual: {e}")
64
65         pending_snapshot = self.predictor.pending_snapshot_future
66         if pending_snapshot is not None and not pending_snapshot.done():
67             try:
68                 remaining_time = max(1, timeout_seconds - (time.time() - shutdown_start))
69                 pending_snapshot.result(timeout=remaining_time)
70             except Exception as e:
71                 self.logger.error(f"Async snapshot failed: {e}")
72
73         try:

```

```

71         final_snapshot_path = f"snapshots/shutdown_{int(time.time())}.pkl"
72         self.predictor.save_snapshot(final_snapshot_path)
73     except Exception as e:
74         self.logger.error(f"Final snapshot failed: {e}")
75
76     try:
77         if hasattr(self.predictor, 'io_executor'):
78             self.predictor.io_executor.shutdown(wait=True, cancel_futures=False)
79     except Exception as e:
80         self.logger.error(f"Error closing resources: {e}")
81
82     total_time = time.time() - shutdown_start
83     self.logger.info(f"SHUTDOWN COMPLETED ({total_time:.2f}s)")
84     sys.exit(0)

```

11 Prometheus Integration

```

1 from prometheus_client import Counter, Histogram
2
3 class UniversalPredictor_GracefulShutdown_Monitored:
4     def __init__(self, config: PredictorConfig):
5         self.shutdown_counter = Counter(
6             'predictor_graceful_shutdowns_total',
7             'Total number of graceful shutdowns executed'
8         )
9         self.shutdown_duration = Histogram(
10            'predictor_shutdown_duration_seconds',
11            'Time taken to complete graceful shutdown',
12            buckets=[0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0]
13        )
14        self.residual_observations = Histogram(
15            'predictor_shutdown_residual_observations',
16            'Number of observations in buffer during shutdown',
17            buckets=[0, 1, 5, 10, 50, 100, 500, 1000]
18        )

```

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

```
1 import logging
2 import os
3 from datetime import datetime
4
5 def save_emergency_dump(predictor, result, asset_id: str):
6     dump_dir = os.path.expanduser("~/predictor_emergency_dumps")
7     os.makedirs(dump_dir, exist_ok=True)
8
9     timestamp = datetime.now().isoformat()
10    dump_file = f"{dump_dir}/{asset_id}_emergency_{timestamp}.msgpack"
11
12    debug_payload = {
13        "emergency_timestamp": timestamp,
14        "asset_id": asset_id,
15        "holder_exponent": float(result.holder_exponent),
16        "weights": [float(w) for w in result.weights],
17        "signal_buffer": predictor._state.signal_circular_buffer.tolist(),
18        "regime_history": predictor._state.cusum_history.tolist(),
19        "telemetry_snapshot": {
20            "kurtosis": float(result.kurtosis),
21            "dgm_entropy": float(result.dgm_entropy),
22            "adaptive_threshold": float(result.adaptive_threshold),
23            "distance_to_collapse": float(result.distance_to_collapse)
24        },
25        "flags_at_emergency": {
26            "degraded_inference": bool(result.degraded_inference_mode),
27            "regime_change": bool(result.regime_change_detected),
28            "mode_collapse": bool(result.mode_collapse_warning)
29        }
30    }
31
32    with open(dump_file, "wb") as f:
33        msgpack.packb(debug_payload, file=f)
34
35    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:

```
1 import os
2 import numpy as np
3 import jax
4
5 os.environ['XLA_FLAGS'] = '--xla_cpu_use_cross_replica_callbacks=false'
6 os.environ['JAX_DETERMINISTIC_REDUCATIONS'] = '1'
7 os.environ['JAX_TRACEBACK_FILTERING'] = 'off'
8
9 np.random.seed(42)
10
11 jax.config.update('jax_default_prng_impl', 'threefry2x32')
12 key = jax.random.PRNGKey(42)
13
14 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.

21 Meta-Optimization API (Bayesian Hyperparameter Tuning)

To support the autonomous Level 4 operation defined in `Stochastic_Predictor_Implementation.tex` (tiered meta-optimization), the API must expose contracts for persistent, resumable Bayesian optimization.

21.1 BayesianMetaOptimizer Class

The `BayesianMetaOptimizer` encapsulates the Tree-structured Parzen Estimator (TPE) logic for both Fast Tuning and Deep Tuning regimes.

```
1 from typing import Callable, Dict, Any, Optional
2 from dataclasses import dataclass
3 import pickle
4 import hashlib
5 from pathlib import Path
6
7 @dataclass(frozen=True)
8 class SearchSpace:
9     """Defines hyperparameter search space with constraints."""
10     name: str
11     param_type: str # "float", "int", "categorical", "log_uniform"
12     range: tuple[Any, Any] # (min, max) or list of choices
13     locked: bool = False # Immutable parameters (security/io sections)
14     constraint: Optional[str] = None # "must be power of 2", etc.
15
16 class BayesianMetaOptimizer:
17     """
18     Resumable Bayesian optimization using TPE algorithm.
19     Supports checkpointing for long-running Deep Tuning campaigns.
20     """
21
22     def __init__(
23         self,
24         search_space: Dict[str, SearchSpace],
25         objective_fn: Callable[[Dict[str, Any]], float],
26         study_name: str,
27         max_iterations: int,
28         tier: str = "fast" # "fast" or "deep"
29     ):
30         """
31         Initialize optimizer with search space and objective function.
32
33         Args:
34             search_space: Dictionary of parameter names to SearchSpace definitions
35             objective_fn: Walk-forward validation function returning MAPE
36             study_name: Unique identifier for this optimization campaign
37             max_iterations: Budget (50 for Fast, 500 for Deep)
38             tier: Optimization tier ("fast" or "deep")
39         """
```

```

40     Raises:
41         ValueError: If search_space contains locked parameters
42     """
43     self.search_space = self._validate_search_space(search_space)
44     self.objective_fn = objective_fn
45     self.study_name = study_name
46     self.max_iterations = max_iterations
47     self.tier = tier
48
49     # Internal TPE state (optuna.Study or similar)
50     self._study = None
51     self._best_params = None
52     self._best_value = float('inf')
53     self._iteration = 0
54     self._checkpoint_counter = 0
55
56     def _validate_search_space(self, space: Dict[str, SearchSpace]) -> Dict[str, SearchSpace]:
57         """Verify no locked parameters in search space."""
58         locked_params = [name for name, spec in space.items() if spec.locked]
59         if locked_params:
60             raise ValueError(
61                 f"Cannot optimize locked parameters: {locked_params}. "
62                 f"Remove from search space or set locked=False."
63             )
64         return space
65
66     def optimize(
67         self,
68         checkpoint_interval: int = 10,
69         early_stopping_patience: int = 50
70     ) -> Dict[str, Any]:
71         """
72         Execute Bayesian optimization with automatic checkpointing.
73
74         Args:
75             checkpoint_interval: Emit checkpoint every N trials
76             early_stopping_patience: Stop if no improvement for N trials
77
78         Returns:
79             Best hyperparameter configuration found
80
81         Note:
82             This method blocks until completion or early stopping.
83             For long-running Deep Tuning, consider running in separate process.
84         """
85         no_improvement_count = 0
86
87         for i in range(self._iteration, self.max_iterations):
88             # Sample next candidate from TPE surrogate
89             candidate = self._suggest_next_candidate()
90
91             # Evaluate via walk-forward validation (expensive!)
92             objective_value = self.objective_fn(candidate)
93
94             # Update TPE model
95             self._report_trial(candidate, objective_value)
96
97             # Track best result
98             if objective_value < self._best_value:
99                 self._best_value = objective_value
100                 self._best_params = candidate
101                 no_improvement_count = 0
102                 # Checkpoint immediately on improvement
103                 self.save_study(f"io/snapshots/{self.study_name}_best.pkl")
104             else:
105                 no_improvement_count += 1
106
107             self._iteration = i + 1
108
109             # Periodic checkpointing
110             if (i + 1) % checkpoint_interval == 0:
111                 checkpoint_path = f"io/snapshots/{self.study_name}_iter{i+1}.pkl"
112                 self.save_study(checkpoint_path)

```

```

113         # Early stopping
114         if no_improvement_count >= early_stopping_patience:
115             print(f"Early stopping: No improvement for {early_stopping_patience} trials")
116             break
117
118     return self._best_params
119
120
121 def save_study(self, path: str) -> None:
122     """
123     Serialize TPE study state to disk for resumability.
124
125     Implementation must guarantee atomic write via temporary file + os.replace().
126     Includes SHA-256 hash for integrity verification on load.
127
128     Args:
129         path: Target checkpoint file path (e.g., "io/snapshots/study.pkl")
130
131     Protocol:
132         1. Serialize study state to temporary file
133         2. Compute SHA-256 hash of serialized data
134         3. Atomically replace target file (POSIX os.replace)
135         4. Store hash in metadata sidecar file
136
137     Note:
138         This operation is I/O-bound and may block for 100-500ms.
139         For production systems running live prediction, execute in
140         separate thread or process to avoid blocking telemetry collection.
141
142     Example:
143         >>> optimizer.save_study("checkpoints/deep_tuning_iter250.pkl")
144     """
145     path_obj = Path(path)
146     path_obj.parent.mkdir(parents=True, exist_ok=True)
147
148     # Prepare checkpoint payload
149     checkpoint_data = {
150         'study_name': self.study_name,
151         'search_space': self.search_space,
152         'tier': self.tier,
153         'iteration': self._iteration,
154         'best_params': self._best_params,
155         'best_value': self._best_value,
156         'trial_history': self._study.trials if self._study else [],
157         'parzen_estimators': self._study._storage if self._study else None,
158         'rng_state': self._get_rng_state(),
159         'timestamp': time.time_ns()
160     }
161
162     # Serialize to temporary file (atomic write protocol)
163     tmp_path = path_obj.with_suffix('.tmp')
164     with open(tmp_path, 'wb') as f:
165         serialized = pickle.dumps(checkpoint_data, protocol=pickle.HIGHEST_PROTOCOL)
166         f.write(serialized)
167         f.flush()
168         os.fsync(f.fileno()) # Force kernel buffer flush
169
170     # Compute integrity hash
171     with open(tmp_path, 'rb') as f:
172         hash_value = hashlib.sha256(f.read()).hexdigest()
173
174     # Atomic replacement (POSIX guarantee)
175     os.replace(tmp_path, path)
176
177     # Store hash in sidecar
178     hash_path = path_obj.with_suffix('.pkl.sha256')
179     with open(hash_path, 'w') as f:
180         f.write(f"{hash_value} {path_obj.name}\n")
181
182 def load_study(self, path: str) -> None:
183     """
184     Deserialize TPE study state from checkpoint.
185

```

```

186     Verifies SHA-256 hash before loading to detect corruption.
187     Reconstructs Parzen estimators and random state for exact resumption.
188
189     Args:
190         path: Checkpoint file path
191
192     Raises:
193         IntegrityError: If SHA-256 verification fails
194         FileNotFoundError: If checkpoint or hash file missing
195         ValueError: If checkpoint schema version incompatible
196
197     Protocol:
198         1. Verify SHA-256 hash matches expected value
199         2. Deserialize checkpoint data
200         3. Reconstruct TPE study object
201         4. Restore RNG state for deterministic sampling
202         5. Validate search space matches current configuration
203
204     Note:
205         After successful load, optimizer continues from iteration N+1.
206         No re-evaluation of previous trials occurs (warm start).
207
208     Example:
209         >>> optimizer = BayesianMetaOptimizer(...)
210         >>> optimizer.load_study("checkpoints/deep_tuning_iter250.pkl")
211         >>> optimizer.optimize() # Resumes from iteration 251
212     """
213     path_obj = Path(path)
214     hash_path = path_obj.with_suffix('.pkl.sha256')
215
216     # Verify integrity
217     if not hash_path.exists():
218         raise FileNotFoundError(f"Hash file missing: {hash_path}")
219
220     with open(hash_path, 'r') as f:
221         expected_hash = f.read().strip().split()[0]
222
223     with open(path, 'rb') as f:
224         actual_hash = hashlib.sha256(f.read()).hexdigest()
225
226     if actual_hash != expected_hash:
227         raise IntegrityError(
228             f"Checkpoint corrupted: hash mismatch. "
229             f"Expected {expected_hash}, got {actual_hash}"
230         )
231
232     # Deserialize
233     with open(path, 'rb') as f:
234         checkpoint_data = pickle.load(f)
235
236     # Validate schema
237     if checkpoint_data['study_name'] != self.study_name:
238         raise ValueError(
239             f"Study name mismatch: checkpoint is for '{checkpoint_data['study_name']}', "
240             f"but optimizer is '{self.study_name}'"
241         )
242
243     # Restore state
244     self._iteration = checkpoint_data['iteration']
245     self._best_params = checkpoint_data['best_params']
246     self._best_value = checkpoint_data['best_value']
247
248     # Reconstruct TPE study (replay trials)
249     self._study = self._create_study()
250     for trial_data in checkpoint_data['trial_history']:
251         self._study.add_trial(trial_data)
252
253     # Restore RNG state for deterministic continuation
254     self._restore_rng_state(checkpoint_data['rng_state'])
255
256     print(f"Resumed from iteration {self._iteration}, best value: {self._best_value:.6f}")

```

21.2 OptimizationResult Schema (Pydantic)

The result of a meta-optimization campaign must be exportable to `config.toml` using the atomic mutation protocol.

```
1 from pydantic import BaseModel, validator
2 from typing import Dict, Any, Optional
3 import toml
4 import os
5 import time
6
7 class OptimizationResult(BaseModel):
8     """
9     Immutable result of Bayesian optimization campaign.
10    Includes export capability for atomic config mutation.
11    """
12    study_name: str
13    tier: str # "fast" or "deep"
14    best_params: Dict[str, Any]
15    best_objective: float
16    total_iterations: int
17    early_stopped: bool
18    convergence_delta: float # Improvement over last N trials
19    timestamp_utc: str
20
21    @validator('tier')
22    def validate_tier(cls, v):
23        if v not in ['fast', 'deep']:
24            raise ValueError(f"Invalid tier: {v}. Must be 'fast' or 'deep'")
25        return v
26
27    def export_to_toml(
28        self,
29        path: str = "config.toml",
30        backup: bool = True,
31        validate: bool = True
32    ) -> None:
33        """
34        Export optimized parameters to config.toml using atomic mutation protocol.
35
36        Implements the Configuration Mutation Protocol specified in
37        Stochastic_Predictor_I0.tex §3.3.
38
39        Args:
40            path: Target config file path (default: "config.toml")
41            backup: Create timestamped backup before mutation (default: True)
42            validate: Validate merged config against schema (default: True)
43
44        Protocol:
45            1. Validate new parameters against schema (ranges, types, constraints)
46            2. Create immutable backup (config.toml.bak + timestamped archive)
47            3. Write to temporary file (config.toml.tmp)
48            4. Fsync to guarantee durability
49            5. Atomic replacement via os.replace()
50            6. Log mutation to audit trail (io/mutations.log)
51
52        Locked Subsections (Never Modified):
53            - [io]: snapshot_path, telemetry_buffer_maxlen, credentials_vault_path
54            - [security]: telemetry_hash_interval_steps, snapshot_integrity_hash_algorithm
55            - [core]: float_precision, jax_platform (partial lock)
56            - [meta_optimization]: max_deep_tuning_iterations, checkpoint_path
57
58        Raises:
59            ConfigMutationError: If validation fails or locked parameter modified
60            IOError: If atomic write fails (disk full, permissions, etc.)
61
62        Note:
63            This operation blocks for ~50-200ms due to fsync requirement.
64            For production systems, execute in separate thread/process.
65
66        Example:
67            >>> result = optimizer.optimize()
68            >>> opt_result = OptimizationResult(
69                ...     study_name="deep_tuning_2026",
```



```

70         tier="deep",
71         best_params=result,
72         best_objective=0.0234,
73         ...
74     )
75     >>> # Non-blocking export in separate thread
76     >>> import threading
77     >>> export_thread = threading.Thread(
78         target=opt_result.export_to_toml,
79         kwargs={'backup': True, 'validate': True}
80     )
81     >>> export_thread.start()
82     """
83     path_obj = Path(path)
84     if not path_obj.exists():
85         raise FileNotFoundError(f"Config file not found: {path}")
86
87     # Phase 1: Load and merge
88     current_config = toml.load(path)
89     merged_config = self._merge_with_validation(current_config, validate)
90
91     # Phase 2: Backup
92     if backup:
93         timestamp = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
94         backup_timestamped = path_obj.with_suffix(f'.bak.{timestamp}')
95         backup_latest = path_obj.with_suffix('.bak')
96
97         import shutil
98         shutil.copy2(path, backup_timestamped)
99         shutil.copy2(path, backup_latest)
100
101     # Phase 3: Atomic write via temporary file
102     tmp_path = path_obj.with_suffix('.tmp')
103
104     # Prevent concurrent mutations
105     if tmp_path.exists():
106         raise IOError(
107             f"Concurrent mutation detected: {tmp_path} exists. "
108             f"Another optimizer may be writing. Aborting."
109         )
110
111     with open(tmp_path, 'w') as f:
112         toml.dump(merged_config, f)
113         f.flush()
114         os.fsync(f.fileno()) # CRITICAL: Force kernel buffer flush
115
116     # Phase 4: Atomic replacement (POSIX guarantee)
117     os.replace(tmp_path, path)
118
119     # Phase 5: Audit logging
120     delta = self._compute_delta(current_config, merged_config)
121     self._log_mutation(delta, timestamp if backup else time.strftime("%Y-%m-%dT%H:%M:%SZ"))
122 )
123
124 def _merge_with_validation(
125     self,
126     current_config: Dict[str, Any],
127     validate: bool
128 ) -> Dict[str, Any]:
129     """
130     Merge optimized parameters into current config with validation.
131
132     Prevents modification of locked subsections.
133     Validates ranges and constraints.
134     """
135     merged = current_config.copy()
136
137     # Determine target subsection based on tier
138     target_section = "sensitivity" if self.tier == "fast" else "structural"
139
140     if target_section not in merged:
141         merged[target_section] = {}

```

```

142     for param_name, param_value in self.best_params.items():
143         # Check if parameter is in locked subsection
144         if self._is_locked_parameter(param_name):
145             raise ConfigMutationError(
146                 f"Attempted to modify locked parameter: {param_name}. "
147                 f"This violates invariant protection rules."
148             )
149
150         merged[target_section][param_name] = param_value
151
152     if validate:
153         self._validate_config(merged)
154
155     return merged
156
157 def _is_locked_parameter(self, param_name: str) -> bool:
158     """Check if parameter belongs to locked subsection."""
159     locked_params = {
160         'snapshot_path', 'telemetry_buffer_maxlen', 'credentials_vault_path',
161         'telemetry_hash_interval_steps', 'snapshot_integrity_hash_algorithm',
162         'float_precision', 'jax_platform',
163         'max_deep_tuning_iterations', 'checkpoint_path'
164     }
165     return param_name in locked_params
166
167 def _validate_config(self, config: Dict[str, Any]) -> None:
168     """Validate merged config against schema."""
169     # Implementation: Check ranges, types, constraints
170     # Raise ConfigMutationError if validation fails
171     pass
172
173 def _compute_delta(
174     self,
175     old_config: Dict[str, Any],
176     new_config: Dict[str, Any]
177 ) -> Dict[str, Any]:
178     """Compute parameter delta for audit logging."""
179     delta = {}
180     # Compare configs and extract changes
181     return delta
182
183 def _log_mutation(self, delta: Dict[str, Any], timestamp: str) -> None:
184     """Append mutation record to audit trail."""
185     log_path = Path("io/mutations.log")
186     log_path.parent.mkdir(parents=True, exist_ok=True)
187
188     with open(log_path, 'a') as f:
189         f.write(f"[{timestamp}] MUTATION_SUCCESS\n")
190         f.write(f"  Trigger: {self.tier.capitalize()}Tuning_{self.study_name}\n")
191         f.write(f"  Best_Objective: {self.best_objective:.6f}\n")
192         f.write(f"  Delta:\n")
193         for param, change in delta.items():
194             f.write(f"    - {param}: {change}\n")
195         f.write("\n")
196
197 class ConfigMutationError(Exception):
198     """Raised when config mutation violates invariant protection rules."""
199     pass
200
201 class IntegrityError(Exception):
202     """Raised when checkpoint integrity verification fails."""
203     pass

```

21.3 Non-Blocking I/O Execution Pattern

Meta-optimization I/O operations (checkpoint save/load, TOML export) are blocking by nature due to `fsync()` requirements. To prevent interference with live prediction and telemetry collection, these operations must execute in separate threads or processes.

```

1 import threading
2 import queue
3 from concurrent.futures import ThreadPoolExecutor

```

```

4
5 class AsyncMetaOptimizer:
6     """
7     Wrapper for BayesianMetaOptimizer with non-blocking I/O.
8     Checkpoints and config exports execute in background threads.
9     """
10
11     def __init__(self, optimizer: BayesianMetaOptimizer):
12         self.optimizer = optimizer
13         self._io_executor = ThreadPoolExecutor(max_workers=2, thread_name_prefix="meta_io")
14         self._checkpoint_queue = queue.Queue(maxsize=5)
15
16     def save_study_async(self, path: str) -> None:
17         """
18         Non-blocking checkpoint save.
19         Submits to thread pool and returns immediately.
20
21         Note:
22             If checkpoint queue is full (5 pending), oldest is dropped (backpressure).
23         """
24         if self._checkpoint_queue.full():
25             # Drop oldest pending checkpoint to prevent memory buildup
26             try:
27                 self._checkpoint_queue.get_nowait()
28             except queue.Empty:
29                 pass
30
31         future = self._io_executor.submit(self.optimizer.save_study, path)
32         self._checkpoint_queue.put(future)
33
34     def export_config_async(
35         self,
36         result: OptimizationResult,
37         path: str = "config.toml"
38     ) -> None:
39         """
40         Non-blocking config export.
41         Returns immediately, actual write happens in background.
42
43         WARNING:
44             The config file mutation happens asynchronously.
45             Do not restart the predictor until export completes.
46             Use wait_for_io_completion() to block until done.
47         """
48         self._io_executor.submit(result.export_to_toml, path=path, backup=True)
49
50     def wait_for_io_completion(self, timeout_seconds: float = 60.0) -> bool:
51         """
52         Block until all pending I/O operations complete.
53
54         Args:
55             timeout_seconds: Maximum wait time
56
57         Returns:
58             True if all operations completed, False if timeout
59
60         Use Case:
61             Before restarting predictor after config mutation:
62             >>> async_opt.export_config_async(result)
63             >>> if async_opt.wait_for_io_completion(timeout_seconds=30):
64             >>>     predictor.reload_config() # Safe to reload
65         """
66         deadline = time.time() + timeout_seconds
67
68         while not self._checkpoint_queue.empty():
69             remaining = deadline - time.time()
70             if remaining <= 0:
71                 return False
72
73             try:
74                 future = self._checkpoint_queue.get(timeout=remaining)
75                 future.result(timeout=remaining) # Wait for completion
76             except queue.Empty:

```

```

77         break
78     except Exception as e:
79         print(f"I/O operation failed: {e}")
80         return False
81
82     # Wait for executor to finish all tasks
83     self._io_executor.shutdown(wait=True, cancel_futures=False)
84     return True
85
86 # Example usage in production
87 if __name__ == "__main__":
88     # Setup optimizer
89     search_space = {
90         'cusum_k': SearchSpace('cusum_k', 'float', (0.3, 1.5)),
91         'dgm_width_size': SearchSpace('dgm_width_size', 'int', (32, 256)),
92         # ... more parameters
93     }
94
95     optimizer = BayesianMetaOptimizer(
96         search_space=search_space,
97         objective_fn=walk_forward_validation,
98         study_name="deep_tuning_2026_Q1",
99         max_iterations=500,
100         tier="deep"
101     )
102
103     # Wrap for non-blocking I/O
104     async_optimizer = AsyncMetaOptimizer(optimizer)
105
106     # Attempt resume from checkpoint
107     checkpoint_path = "io/snapshots/deep_tuning_2026_Q1_iter250.pkl"
108     if Path(checkpoint_path).exists():
109         optimizer.load_study(checkpoint_path) # Blocking load is OK (one-time startup)
110
111     # Run optimization with non-blocking checkpoints
112     for i in range(optimizer._iteration, optimizer.max_iterations):
113         candidate = optimizer._suggest_next_candidate()
114         objective = walk_forward_validation(candidate)
115         optimizer._report_trial(candidate, objective)
116
117         if (i + 1) % 10 == 0:
118             # Non-blocking checkpoint (doesn't interrupt optimization loop)
119             checkpoint = f"io/snapshots/deep_tuning_2026_Q1_iter{i+1}.pkl"
120             async_optimizer.save_study_async(checkpoint)
121
122     # Export results to config.toml (non-blocking)
123     result = OptimizationResult(
124         study_name="deep_tuning_2026_Q1",
125         tier="deep",
126         best_params=optimizer._best_params,
127         best_objective=optimizer._best_value,
128         total_iterations=optimizer._iteration,
129         early_stopped=False,
130         convergence_delta=0.0001,
131         timestamp_utc=time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
132     )
133     async_optimizer.export_config_async(result)
134
135     # Wait for all I/O to complete before exiting
136     if async_optimizer.wait_for_io_completion(timeout_seconds=60):
137         print("All I/O operations completed successfully")
138     else:
139         print("WARNING: Some I/O operations timed out")

```

21.4 Integration with Prediction Pipeline

Meta-optimization runs offline (batch mode) separate from live prediction. However, config mutations can occur during production operation. The predictor must detect config changes and reload safely.

```

1 class ConfigReloadablePredictor(UniversalPredictor):
2     """
3     UniversalPredictor with hot-reload capability for config mutations.

```

```

4      """
5
6      def __init__(self, config_path: str = "config.toml"):
7          self.config_path = config_path
8          self._config_mtime = os.path.getmtime(config_path)
9
10         config = self._load_config(config_path)
11         super().__init__(config)
12
13     def check_and_reload_config(self) -> bool:
14         """
15         Check if config.toml has been modified and reload if necessary.
16
17         Returns:
18             True if config was reloaded, False if unchanged
19
20         Note:
21             Reloading config triggers full state reinitialization.
22             Call this only during safe windows (e.g., market closed, low traffic).
23         """
24         current_mtime = os.path.getmtime(self.config_path)
25
26         if current_mtime > self._config_mtime:
27             print(f"Config file modified. Reloading from {self.config_path}")
28             new_config = self._load_config(self.config_path)
29
30             # Reinitialize with new config
31             self.config = new_config
32             self._state = self._initialize_state()
33             self._jit_update = jax.jit(self._core_update_step)
34
35             self._config_mtime = current_mtime
36             return True
37
38     return False

```

22 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.