Python API Specification - Universal Predictor

Software Engineering

February 20, 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
    jko_domain_length: float = 1.0  # Domain length for JKO scaling
    entropy_window_relaxation_factor: float = 5.0  # Relaxation multiplier
    entropy_window_bounds_min: int = 10  # Minimum entropy window
    entropy_window_bounds_max: int = 500  # Maximum entropy window
    learning_rate_safety_factor: float = 0.8  # Safety factor for learning rate
    learning_rate_minimum: float = 1e-6  # Minimum 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
    koopman_top_k: int = 5         # Top-K Koopman spectral modes
    koopman_min_power: float = 1e-10  # Minimum Koopman spectral power
    paley_wiener_integral_max: float = 100.0  # Paley-Wiener threshold
    kernel_c_jump_intensity: float = 0.05  # Levy jump intensity
    kernel_c_jump_mean: float = 0.0  # Levy jump mean
    kernel_c_jump_scale: float = 0.1  # Levy jump scale
    kernel_c_jump_max_events: int = 16  # Max jump events per step
    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
    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
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

```
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:
            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

```
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.

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.

```python
from typing import Callable, Dict, Any, Optional
from dataclasses import dataclass
import pickle
import hashlib
from pathlib import Path

@dataclass(frozen=True)
class SearchSpace:
    """Defines hyperparameter search space with constraints."""
    name: str
    param_type: str  # "float", "int", "categorical", "log_uniform"
    range: tuple[Any, Any]  # (min, max) or list of choices
    locked: bool = False  # Immutable parameters (security/io sections)
    constraint: Optional[str] = None  # "must be power of 2", etc.

class BayesianMetaOptimizer:
    """
    Resumable Bayesian optimization using TPE algorithm.
    Supports checkpointing for long-running Deep Tuning campaigns.
    """

    def __init__(
        self,
        search_space: Dict[str, SearchSpace],
        objective_fn: Callable[[Dict[str, Any]], float],
        study_name: str,
        max_iterations: int,
        tier: str = "fast"  # "fast" or "deep"
    ):
        """
        Initialize optimizer with search space and objective function.

        Args:
            search_space: Dictionary of parameter names to SearchSpace definitions
            objective_fn: Walk-forward validation function returning MAPE
            study_name: Unique identifier for this optimization campaign
            max_iterations: Budget (50 for Fast, 500 for Deep)
            tier: Optimization tier ("fast" or "deep")

        Raises:
            ValueError: If search_space contains locked parameters
        """
        self.search_space = self._validate_search_space(search_space)
        self.objective_fn = objective_fn
        self.study_name = study_name
        self.max_iterations = max_iterations
        self.tier = tier

        # Internal TPE state (optuna.Study or similar)
        self._study = None
        self._best_params = None
        self._best_value = float('inf')
        self._iteration = 0
        self._checkpoint_counter = 0

    def _validate_search_space(self, space: Dict[str, SearchSpace]) -> Dict[str, SearchSpace]:
        """Verify no locked parameters in search space."""
        locked_params = [name for name, spec in space.items() if spec.locked]
        if locked_params:
            raise ValueError(
                f"Cannot optimize locked parameters: {locked_params}. "
                f"Remove from search space or set locked=False."
            )
        return space

    def optimize(
        self,
        checkpoint_interval: int = 10,
        early_stopping_patience: int = 50
    ) -> Dict[str, Any]:
        """
        Execute Bayesian optimization with automatic checkpointing.

        Args:
            checkpoint_interval: Emit checkpoint every N trials
            early_stopping_patience: Stop if no improvement for N trials

        Returns:
            Best hyperparameter configuration found

        Note:
            This method blocks until completion or early stopping.
            For long-running Deep Tuning, consider running in separate process.
        """
        no_improvement_count = 0

        for i in range(self._iteration, self.max_iterations):
            # Sample next candidate from TPE surrogate
            candidate = self._suggest_next_candidate()

            # Evaluate via walk-forward validation (expensive!)
            objective_value = self.objective_fn(candidate)

            # Update TPE model
            self._report_trial(candidate, objective_value)

            # Track best result
            if objective_value < self._best_value:
                self._best_value = objective_value
                self._best_params = candidate
                no_improvement_count = 0
                # Checkpoint immediately on improvement
                self.save_study(f"io/snapshots/{self.study_name}_best.pkl")
            else:
                no_improvement_count += 1

            self._iteration = i + 1

            # Periodic checkpointing
            if (i + 1) % checkpoint_interval == 0:
                checkpoint_path = f"io/snapshots/{self.study_name}_iter{i+1}.pkl"
                self.save_study(checkpoint_path)

            # Early stopping
            if no_improvement_count >= early_stopping_patience:
                print(f"Early stopping: No improvement for {early_stopping_patience} trials")
                break

        return self._best_params

    def save_study(self, path: str) -> None:
        """
        Serialize TPE study state to disk for resumability.

        Implementation must guarantee atomic write via temporary file + os.replace().
        Includes SHA-256 hash for integrity verification on load.

        Args:
            path: Target checkpoint file path (e.g., "io/snapshots/study.pkl")

        Protocol:
            1. Serialize study state to temporary file
            2. Compute SHA-256 hash of serialized data
            3. Atomically replace target file (POSIX os.replace)
            4. Store hash in metadata sidecar file

        Note:
            This operation is I/O-bound and may block for 100-500ms.
            For production systems running live prediction, execute in
            separate thread or process to avoid blocking telemetry collection.

        Example:
            >>> optimizer.save_study("checkpoints/deep_tuning_iter250.pkl")
        """
        path_obj = Path(path)
        path_obj.parent.mkdir(parents=True, exist_ok=True)

        # Prepare checkpoint payload
        checkpoint_data = {
            'study_name': self.study_name,
            'search_space': self.search_space,
            'tier': self.tier,
            'iteration': self._iteration,
            'best_params': self._best_params,
            'best_value': self._best_value,
            'trial_history': self._study.trials if self._study else [],
            'parzen_estimators': self._study._storage if self._study else None,
            'rng_state': self._get_rng_state(),
            'timestamp': time.time_ns()
        }

        # Serialize to temporary file (atomic write protocol)
        tmp_path = path_obj.with_suffix('.tmp')
        with open(tmp_path, 'wb') as f:
            serialized = pickle.dumps(checkpoint_data, protocol=pickle.HIGHEST_PROTOCOL)
            f.write(serialized)
            f.flush()
            os.fsync(f.fileno())  # Force kernel buffer flush

        # Compute integrity hash
        with open(tmp_path, 'rb') as f:
            hash_value = hashlib.sha256(f.read()).hexdigest()

        # Atomic replacement (POSIX guarantee)
        os.replace(tmp_path, path)

        # Store hash in sidecar
        hash_path = path_obj.with_suffix('.pkl.sha256')
        with open(hash_path, 'w') as f:
            f.write(f"{hash_value}  {path_obj.name}\n")

    def load_study(self, path: str) -> None:
        """
        Deserialize TPE study state from checkpoint.

        Verifies SHA-256 hash before loading to detect corruption.
        Reconstructs Parzen estimators and random state for exact resumption.

        Args:
            path: Checkpoint file path

        Raises:
            IntegrityError: If SHA-256 verification fails
            FileNotFoundError: If checkpoint or hash file missing
            ValueError: If checkpoint schema version incompatible

        Protocol:
            1. Verify SHA-256 hash matches expected value
            2. Deserialize checkpoint data
            3. Reconstruct TPE study object
            4. Restore RNG state for deterministic sampling
            5. Validate search space matches current configuration

        Note:
            After successful load, optimizer continues from iteration N+1.
            No re-evaluation of previous trials occurs (warm start).

        Example:
            >>> optimizer = BayesianMetaOptimizer(...)
            >>> optimizer.load_study("checkpoints/deep_tuning_iter250.pkl")
            >>> optimizer.optimize()  # Resumes from iteration 251
        """
        path_obj = Path(path)
        hash_path = path_obj.with_suffix('.pkl.sha256')

        # Verify integrity
        if not hash_path.exists():
            raise FileNotFoundError(f"Hash file missing: {hash_path}")

        with open(hash_path, 'r') as f:
            expected_hash = f.read().strip().split()[0]

        with open(path, 'rb') as f:
            actual_hash = hashlib.sha256(f.read()).hexdigest()

        if actual_hash != expected_hash:
            raise IntegrityError(
                f"Checkpoint corrupted: hash mismatch. "
                f"Expected {expected_hash}, got {actual_hash}"
            )

        # Deserialize
        with open(path, 'rb') as f:
            checkpoint_data = pickle.load(f)

        # Validate schema
        if checkpoint_data['study_name'] != self.study_name:
            raise ValueError(
```

```python
                f"Study name mismatch: checkpoint is for '{checkpoint_data['study_name']}', "
                f"but optimizer is '{self.study_name}'"
            )

        # Restore state
        self._iteration = checkpoint_data['iteration']
        self._best_params = checkpoint_data['best_params']
        self._best_value = checkpoint_data['best_value']

        # Reconstruct TPE study (replay trials)
        self._study = self._create_study()
        for trial_data in checkpoint_data['trial_history']:
            self._study.add_trial(trial_data)

        # Restore RNG state for deterministic continuation
        self._restore_rng_state(checkpoint_data['rng_state'])

        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.

```python
from pydantic import BaseModel, validator
from typing import Dict, Any, Optional
import toml
import os
import time

class OptimizationResult(BaseModel):
    """
    Immutable result of Bayesian optimization campaign.
    Includes export capability for atomic config mutation.
    """
    study_name: str
    tier: str  # "fast" or "deep"
    best_params: Dict[str, Any]
    best_objective: float
    total_iterations: int
    early_stopped: bool
    convergence_delta: float  # Improvement over last N trials
    timestamp_utc: str

    @validator('tier')
    def validate_tier(cls, v):
        if v not in ['fast', 'deep']:
            raise ValueError(f"Invalid tier: {v}. Must be 'fast' or 'deep'")
        return v

    def export_to_toml(
        self,
        path: str = "config.toml",
        backup: bool = True,
        validate: bool = True
    ) -> None:
        """
        Export optimized parameters to config.toml using atomic mutation protocol.

        Implements the Configuration Mutation Protocol specified in
        Stochastic_Predictor_IO.tex §3.3.

        Args:
            path: Target config file path (default: "config.toml")
            backup: Create timestamped backup before mutation (default: True)
            validate: Validate merged config against schema (default: True)

        Protocol:
            1. Validate new parameters against schema (ranges, types, constraints)
            2. Create immutable backup (config.toml.bak + timestamped archive)
            3. Write to temporary file (config.toml.tmp)
            4. Fsync to guarantee durability
            5. Atomic replacement via os.replace()
            6. Log mutation to audit trail (io/mutations.log)

        Locked Subsections (Never Modified):
            - [io]: snapshot_path, telemetry_buffer_maxlen, credentials_vault_path
            - [security]: telemetry_hash_interval_steps, snapshot_integrity_hash_algorithm
            - [core]: float_precision, jax_platform (partial lock)
            - [meta_optimization]: max_deep_tuning_iterations, checkpoint_path

        Raises:
            ConfigMutationError: If validation fails or locked parameter modified
            IOError: If atomic write fails (disk full, permissions, etc.)

        Note:
            This operation blocks for ~50-200ms due to fsync requirement.
            For production systems, execute in separate thread/process.

        Example:
            >>> result = optimizer.optimize()
            >>> opt_result = OptimizationResult(
            ...     study_name="deep_tuning_2026",
            ...     tier="deep",
            ...     best_params=result,
            ...     best_objective=0.0234,
            ...     ...
            ... )
            >>> # Non-blocking export in separate thread
            >>> import threading
            >>> export_thread = threading.Thread(
            ...     target=opt_result.export_to_toml,
            ...     kwargs={'backup': True, 'validate': True}
            ... )
            >>> export_thread.start()
        """
        path_obj = Path(path)
        if not path_obj.exists():
            raise FileNotFoundError(f"Config file not found: {path}")

        # Phase 1: Load and merge
        current_config = toml.load(path)
        merged_config = self._merge_with_validation(current_config, validate)

        # Phase 2: Backup
        if backup:
            timestamp = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
            backup_timestamped = path_obj.with_suffix(f'.bak.{timestamp}')
            backup_latest = path_obj.with_suffix('.bak')

            import shutil
            shutil.copy2(path, backup_timestamped)
            shutil.copy2(path, backup_latest)

        # Phase 3: Atomic write via temporary file
        tmp_path = path_obj.with_suffix('.tmp')

        # Prevent concurrent mutations
        if tmp_path.exists():
            raise IOError(
                f"Concurrent mutation detected: {tmp_path} exists. "
                f"Another optimizer may be writing. Aborting."
            )

        with open(tmp_path, 'w') as f:
            toml.dump(merged_config, f)
            f.flush()
            os.fsync(f.fileno())  # CRITICAL: Force kernel buffer flush

        # Phase 4: Atomic replacement (POSIX guarantee)
        os.replace(tmp_path, path)

        # Phase 5: Audit logging
        delta = self._compute_delta(current_config, merged_config)
        self._log_mutation(delta, timestamp if backup else time.strftime("%Y-%m-%dT%H:%M:%SZ")
    )

    def _merge_with_validation(
        self,
        current_config: Dict[str, Any],
        validate: bool
    ) -> Dict[str, Any]:
        """
        Merge optimized parameters into current config with validation.

        Prevents modification of locked subsections.
        Validates ranges and constraints.
        """
        merged = current_config.copy()

        # Determine target subsection based on tier
        target_section = "sensitivity" if self.tier == "fast" else "structural"

        if target_section not in merged:
            merged[target_section] = {}

        for param_name, param_value in self.best_params.items():
            # Check if parameter is in locked subsection
            if self._is_locked_parameter(param_name):
                raise ConfigMutationError(
                    f"Attempted to modify locked parameter: {param_name}. "
                    f"This violates invariant protection rules."
                )

            merged[target_section][param_name] = param_value

        if validate:
            self._validate_config(merged)

        return merged

    def _is_locked_parameter(self, param_name: str) -> bool:
        """Check if parameter belongs to locked subsection."""
        locked_params = {
            'snapshot_path', 'telemetry_buffer_maxlen', 'credentials_vault_path',
            'telemetry_hash_interval_steps', 'snapshot_integrity_hash_algorithm',
            'float_precision', 'jax_platform',
            'max_deep_tuning_iterations', 'checkpoint_path'
        }
        return param_name in locked_params

    def _validate_config(self, config: Dict[str, Any]) -> None:
        """Validate merged config against schema."""
        # Implementation: Check ranges, types, constraints
        # Raise ConfigMutationError if validation fails
        pass

    def _compute_delta(
        self,
        old_config: Dict[str, Any],
        new_config: Dict[str, Any]
    ) -> Dict[str, Any]:
        """Compute parameter delta for audit logging."""
        delta = {}
        # Compare configs and extract changes
        return delta

    def _log_mutation(self, delta: Dict[str, Any], timestamp: str) -> None:
        """Append mutation record to audit trail."""
        log_path = Path("io/mutations.log")
        log_path.parent.mkdir(parents=True, exist_ok=True)

        with open(log_path, 'a') as f:
            f.write(f"[{timestamp}] MUTATION_SUCCESS\n")
            f.write(f"  Trigger: {self.tier.capitalize()}Tuning_{self.study_name}\n")
            f.write(f"  Best_Objective: {self.best_objective:.6f}\n")
            f.write(f"  Delta:\n")
```

```python
            for param, change in delta.items():
                f.write(f"    - {param}: {change}\n")
            f.write("\n")

class ConfigMutationError(Exception):
    """Raised when config mutation violates invariant protection rules."""
    pass

class IntegrityError(Exception):
    """Raised when checkpoint integrity verification fails."""
    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.

```python
import threading
import queue
from concurrent.futures import ThreadPoolExecutor

class AsyncMetaOptimizer:
    """
    Wrapper for BayesianMetaOptimizer with non-blocking I/O.
    Checkpoints and config exports execute in background threads.
    """

    def __init__(self, optimizer: BayesianMetaOptimizer):
        self.optimizer = optimizer
        self._io_executor = ThreadPoolExecutor(max_workers=2, thread_name_prefix="meta_io")
        self._checkpoint_queue = queue.Queue(maxsize=5)

    def save_study_async(self, path: str) -> None:
        """
        Non-blocking checkpoint save.
        Submits to thread pool and returns immediately.

        Note:
            If checkpoint queue is full (5 pending), oldest is dropped (backpressure).
        """
        if self._checkpoint_queue.full():
            # Drop oldest pending checkpoint to prevent memory buildup
            try:
                self._checkpoint_queue.get_nowait()
            except queue.Empty:
                pass

        future = self._io_executor.submit(self.optimizer.save_study, path)
        self._checkpoint_queue.put(future)

    def export_config_async(
        self,
        result: OptimizationResult,
        path: str = "config.toml"
    ) -> None:
        """
        Non-blocking config export.
        Returns immediately, actual write happens in background.

        WARNING:
            The config file mutation happens asynchronously.
            Do not restart the predictor until export completes.
            Use wait_for_io_completion() to block until done.
        """
        self._io_executor.submit(result.export_to_toml, path=path, backup=True)

    def wait_for_io_completion(self, timeout_seconds: float = 60.0) -> bool:
        """
        Block until all pending I/O operations complete.

        Args:
            timeout_seconds: Maximum wait time

        Returns:
            True if all operations completed, False if timeout

        Use Case:
            Before restarting predictor after config mutation:
            >>> async_opt.export_config_async(result)
            >>> if async_opt.wait_for_io_completion(timeout_seconds=30):
            >>>     predictor.reload_config()  # Safe to reload
        """
        deadline = time.time() + timeout_seconds

        while not self._checkpoint_queue.empty():
            remaining = deadline - time.time()
            if remaining <= 0:
                return False

            try:
                future = self._checkpoint_queue.get(timeout=remaining)
                future.result(timeout=remaining)  # Wait for completion
            except queue.Empty:
                break
            except Exception as e:
                print(f"I/O operation failed: {e}")
                return False

        # Wait for executor to finish all tasks
        self._io_executor.shutdown(wait=True, cancel_futures=False)
        return True

# Example usage in production
if __name__ == "__main__":
    # Setup optimizer
    search_space = {
        'cusum_k': SearchSpace('cusum_k', 'float', (0.3, 1.5)),
        'dgm_width_size': SearchSpace('dgm_width_size', 'int', (32, 256)),
        # ... more parameters
    }

    optimizer = BayesianMetaOptimizer(
        search_space=search_space,
        objective_fn=walk_forward_validation,
        study_name="deep_tuning_2026_Q1",
        max_iterations=500,
        tier="deep"
    )

    # Wrap for non-blocking I/O
    async_optimizer = AsyncMetaOptimizer(optimizer)

    # Attempt resume from checkpoint
    checkpoint_path = "io/snapshots/deep_tuning_2026_Q1_iter250.pkl"
    if Path(checkpoint_path).exists():
        optimizer.load_study(checkpoint_path)  # Blocking load is OK (one-time startup)

    # Run optimization with non-blocking checkpoints
    for i in range(optimizer._iteration, optimizer.max_iterations):
        candidate = optimizer._suggest_next_candidate()
        objective = walk_forward_validation(candidate)
        optimizer._report_trial(candidate, objective)

        if (i + 1) % 10 == 0:
            # Non-blocking checkpoint (doesn't interrupt optimization loop)
            checkpoint = f"io/snapshots/deep_tuning_2026_Q1_iter{i+1}.pkl"
            async_optimizer.save_study_async(checkpoint)

    # Export results to config.toml (non-blocking)
    result = OptimizationResult(
        study_name="deep_tuning_2026_Q1",
        tier="deep",
        best_params=optimizer._best_params,
        best_objective=optimizer._best_value,
        total_iterations=optimizer._iteration,
        early_stopped=False,
        convergence_delta=0.0001,
        timestamp_utc=time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
    )
    async_optimizer.export_config_async(result)

    # Wait for all I/O to complete before exiting
    if async_optimizer.wait_for_io_completion(timeout_seconds=60):
        print("All I/O operations completed successfully")
    else:
        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.

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

    def __init__(self, config_path: str = "config.toml"):
        self.config_path = config_path
        self._config_mtime = os.path.getmtime(config_path)

        config = self._load_config(config_path)
        super().__init__(config)

    def check_and_reload_config(self) -> bool:
        """
        Check if config.toml has been modified and reload if necessary.

        Returns:
            True if config was reloaded, False if unchanged

        Note:
            Reloading config triggers full state reinitialization.
            Call this only during safe windows (e.g., market closed, low traffic).
        """
        current_mtime = os.path.getmtime(self.config_path)

        if current_mtime > self._config_mtime:
            print(f"Config file modified. Reloading from {self.config_path}")
            new_config = self._load_config(self.config_path)

            # Reinitialize with new config
            self.config = new_config
            self._state = self._initialize_state()
            self._jit_update = jax.jit(self._core_update_step)

            self._config_mtime = current_mtime
            return True

        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.