Universal Stochastic Predictor
Phase 4: IO Layer
v2.1.0 (Level 4 Autonomy)

Implementation Team

February 19, 2026

Contents

1 Phase 4: IO Layer Overview **2**
1.1 Tag Information . . . 2
1.2 Scope . . . 2
1.3 Quality Assurance Alignment . . . 2
1.4 Design Principles . . . 2

2 Ingestion and Validation **4**
2.1 Implementation Modules . . . 4
2.2 Catastrophic Outlier Filter . . . 4
2.2.1 Implementation Notes . . . 4
2.3 Frozen Signal Alarm . . . 4
2.3.1 Recovery Criteria (V-MAJ-6: Frozen Signal Recovery Ratio) . . . 5
2.4 Staleness Policy (TTL) . . . 6
2.4.1 Implementation Notes . . . 6

3 Telemetry Abstraction **7**
3.1 TelemetryBuffer Emission . . . 7
3.1.1 Implementation Notes . . . 7
3.2 No Compute Stalls . . . 7

4 Deterministic Logging **8**
4.1 Hash-Based Parity Checks . . . 8
4.2 Audit Hashes . . . 8

5 Snapshot Strategy **9**
5.1 Atomic Persistence . . . 9
5.2 Binary Serialization . . . 9
5.2.1 Implementation Notes . . . 9
5.3 Integrity Verification . . . 9
5.4 Atomic Write Protocol . . . 9
5.4.1 Implementation Notes . . . 10

6 Security Policies **11**
6.1 Credential Injection . . . 11
6.1.1 Implementation Notes . . . 11
6.2 Version Control Exclusion . . . 11

7 Orchestrator Integration **12**
7.1 Ingestion Gate in orchestrate_step() . . . 12
7.1.1 Execution Flow . . . 12
7.1.2 Flag Semantics . . . 13
7.1.3 Early Return on Rejection . . . 13

7.2 PRNG Configuration . . . 13
7.3 64-bit Precision Enforcement . . . 13

8 Telemetry Buffer Integration (P2.3) 14
8.1 Motivation . . . 14
8.2 Data Model . . . 14
8.3 Integration into orchestrate_step() . . . 15
8.4 Configuration Parameters . . . 15
8.4.1 Configuration Injection Example . . . 15
8.5 Thread-Safety Model . . . 16
8.6 Backward Compatibility . . . 16
8.7 Usage Example . . . 16
8.8 Benefits . . . 17
8.9 Visualization Dashboard (GAP-6) . . . 17
8.9.1 Usage . . . 17
8.9.2 Notes . . . 17

9 Level 4 Autonomy: Configuration Mutation Safety 18
9.1 Overview . . . 18
9.2 V-CRIT-2: Atomic Configuration Mutation Protocol . . . 18
9.2.1 Problem Statement . . . 18
9.2.2 Requirements . . . 19
9.2.3 Implementation . . . 19
9.2.4 Usage Example . . . 22
9.2.5 Files Modified . . . 22
9.2.6 Compliance Impact . . . 22
9.3 V-MAJ-4: Configuration Mutation Rate Limiting . . . 23
9.3.1 Problem Statement . . . 23
9.3.2 Safety Requirements . . . 23
9.3.3 Implementation . . . 23
9.3.4 Usage Pattern . . . 24
9.4 V-MAJ-5: Degradation Detection Auto-Rollback . . . 25
9.4.1 Problem Statement . . . 25
9.4.2 Safety Requirements . . . 25
9.4.3 Implementation . . . 25
9.4.4 Usage Pattern . . . 27
9.4.5 Audit Log Format . . . 28
9.5 V-MAJ-7: Adaptive Telemetry Monitoring . . . 28
9.5.1 Implementation Status . . . 28
9.5.2 InternalState Extensions . . . 28
9.5.3 Adaptive Telemetry Collection . . . 28
9.5.4 Integration Pattern . . . 29
9.5.5 Metrics Tracked . . . 30
9.5.6 Future Extensions . . . 30
9.6 Utility Functions . . . 31
9.7 Implementation Status . . . 31

10 Compliance Checklist 32

11 Phase 4 Summary 33

Chapter 1

Phase 4: IO Layer Overview

1.1 Tag Information

- **Tag**: `impl/v2.1.0`
- **Commit**: `6ccb68d` (GAP-6 dashboard implementation + e929bc5 final alignment)
- **Status**: Level 4 Autonomy compliance (V-MAJ-4, V-MAJ-5, V-MAJ-7 implemented; GAP-6 complete)

Phase 4 introduces the asynchronous I/O layer for snapshots, streaming, and telemetry export. The primary design goal is to preserve JAX/XLA throughput by decoupling compute from disk or network latency.

1.2 Scope

Phase 4 covers:

- **Telemetry Buffering**: Non-blocking emission of telemetry snapshots
- **Deterministic Logging**: Hash-based parity checks for CPU/GPU validation
- **Snapshot Strategy**: Atomic persistence of predictor state
- **Ingestion and Validation**: Input filtering, staleness policy, frozen signal detection
- **Security Enforcement**: Credential injection and secret exclusion
- **IO Modules**: validators, loaders, telemetry, snapshots, credentials

1.3 Quality Assurance Alignment

This phase is validated via the project test framework in `Test/`. Automated checks include `flake8`, `black`, `isort`, and `mypy`, plus dependency validation that compares imports to requirements and the active virtual environment. Auto-generated smoke tests are synchronized by `Test/framework/generator.py`. See `doc/latex/specification/Stochastic_Predictor_Tests_Python.tex` for details.

1.4 Design Principles

- **No Compute Stalls**: JAX compute threads never block on I/O
- **Determinism**: Logs capture reproducible hashes instead of raw state dumps

- **Security**: No raw signals or secrets in logs
- **Configurability**: Logging intervals and destinations injected via config
- **Integrity**: Snapshots and parity logs are hash-verified

Chapter 2

Ingestion and Validation

2.1 Implementation Modules

Phase 4 IO introduces the following modules:

- `io/validators.py`: Outlier, frozen signal, and staleness checks
- `io/loaders.py`: Ingestion gate and decision flags
- `io/telemetry.py`: Non-blocking telemetry buffer and parity hashes
- `io/snapshots.py`: Binary snapshots, hash verification, atomic writes
- `io/credentials.py`: Environment-based credential injection helpers

2.2 Catastrophic Outlier Filter

Input validation must reject catastrophic outliers when $|y_t| > 20\sigma$ relative to historical normalization. In this case, the system must preserve inertial state and emit a critical alert without advancing the transport update.

- Reject observation and keep current state unchanged.
- Emit a critical alert for audit visibility.
- Do not update JKO/Sinkhorn weights for the rejected step.

2.2.1 Implementation Notes

Outlier detection is implemented as a pure function with configuration-driven thresholds. The ingestion gate returns a decision object that preserves inertial state when an outlier is detected.

2.3 Frozen Signal Alarm

If the exact same value is observed for $N_{freeze} \geq 5$ consecutive steps, emit a `FrozenSignalAlarmEvent`. This invalidates the multifractal spectrum and requires:

- Freeze the topological branch (Kernel D).
- Switch to degraded inference mode.
- Continue monitoring until signal variation resumes.

2.3.1 Recovery Criteria (V-MAJ-6: Frozen Signal Recovery Ratio)

The frozen signal lock is released when variance recovers above a configurable ratio of historical variance for a configurable number of consecutive steps.

Algorithm

$$\text{recovered} = \text{detect\_frozen\_recovery}(\text{variance\_history}, \text{historical\_var}, \rho, n_c) \tag{2.1}$$

where:

- variance_history: Recent residual variances
- historical_var: Baseline variance reference
- ρ = `config.frozen_signal_recovery_ratio` (default: 0.1): Recovery threshold multiplier
- n_c = `config.frozen_signal_recovery_steps` (default: 2): Confirmation window

Recovery is confirmed when:

$$\text{variance}_t > \rho \cdot \text{historical\_var} \quad \text{for} \quad n_c \text{ consecutive steps} \tag{2.2}$$

Implementation

```
# In evaluate_ingestion():
if frozen:
    residual_window = np.asarray(state.residual_window, dtype=np.float64)
    historical_variance = float(np.var(residual_window)) if residual_window.size > 1 else
     0.0
    recent_window = residual_window[-config.frozen_signal_recovery_steps:]
    recent_variance = float(np.var(recent_window)) if recent_window.size > 1 else 0.0
    variance_history = [recent_variance] * config.frozen_signal_recovery_steps
    in_recovery = detect_frozen_recovery(
        variance_history=variance_history,
        historical_variance=historical_variance,
        ratio_threshold=config.frozen_signal_recovery_ratio,  # V-MAJ-6: Use parameter
        consecutive_steps=config.frozen_signal_recovery_steps
    )
    if in_recovery:
        frozen = False  # Lift the frozen signal flag
```

Configuration Parameters

From `PredictorConfig`:

Parameter	Default	Purpose
`frozen_signal_min_steps`	5	Consecutive equal values to trigger alarm
`frozen_signal_recovery_ratio`	0.1	Variance ratio threshold for recovery (10% of baseline)
`frozen_signal_recovery_steps`	2	Confirmation window for recovery

Table 2.1: V-MAJ-6 Frozen Signal Recovery Configuration

Benefits

- **Automatic Recovery**: No manual intervention needed when signal variance improves
- **Hysteresis**: Recovery threshold ($\rho = 0.1$) is more lenient than typical alarm threshold, preventing oscillation
- **Configuration-Driven**: All parameters injected from config.toml (zero-heuristics policy)
- **State Preservation**: Maintains frozen flag during low-variance periods, automatically lifts when variance returns
- **Signal Quality Supervision**: Enables secondary observability on signal quality degradation patterns

2.4 Staleness Policy (TTL)

Every observation must carry a timestamp for TTL evaluation. If the target delay exceeds Δ_{max}, the JKO update must be suspended immediately.

- Compute staleness as $\Delta_t = t_{now} - t_{obs}$.
- If $\Delta_t > \Delta_{max}$, skip the transport update.
- Preserve state and record a staleness warning event.

2.4.1 Implementation Notes

Staleness is computed as the difference between current time and observation timestamp. The ingestion decision flags a suspended JKO update when the TTL is exceeded. The orchestrator also applies a secondary staleness check using $\Delta t = t_{obs} - t_{last\_update}$ (`timestamp_ns` minus `state.last_update_ns`) to trigger degraded mode on large inter-sample gaps.

Chapter 3

Telemetry Abstraction

3.1 TelemetryBuffer Emission

The JKO orchestrator should emit a `TelemetryBuffer` at the end of each step. This buffer is consumed by a dedicated process outside the JAX execution thread.

- The buffer contains summary metrics (prediction, weights, free energy, diagnostics, regime flags).
- The compute path only enqueues the buffer and continues.
- The consumer is responsible for serialization and persistence.

3.1.1 Implementation Notes

The telemetry buffer is a bounded, thread-safe queue. Buffer capacity is explicitly injected from `PredictorConfig.telemetry_buffer_capacity` to eliminate implicit defaults (zero-heuristics policy). Parity hashes are emitted on a configurable interval and derived from canonical float64 serialization.

```
# Instantiation pattern (capacity injected from config)
buffer = TelemetryBuffer(capacity=config.telemetry_buffer_capacity)
```

3.2 No Compute Stalls

JAX compute threads must never block on I/O. Telemetry buffers must be non-blocking and consumed by a separate process or thread outside the JAX execution path.

Chapter 4

Deterministic Logging

4.1 Hash-Based Parity Checks

For hardware parity audits, the logger records SHA-256 hashes of the weight vector ρ and the OT cost at configurable intervals. This permits CPU/GPU parity validation without dumping VRAM data.

- Hash interval configured per deployment.
- Hashes derived from canonical float64 serialization.
- Logs are append-only and immutable.

4.2 Audit Hashes

Parity audits must log SHA-256 hashes of ρ and OT cost at configured intervals. Hash input must be derived from canonical float64 serialization to ensure reproducibility across CPU and GPU.

Chapter 5

Snapshot Strategy

5.1 Atomic Persistence

Snapshots must be persisted atomically to prevent partial writes. The IO layer is responsible for:

- Writing to temporary files and renaming atomically.
- Optional compression configured by policy.
- Coordinating snapshot cadence with telemetry output.

5.2 Binary Serialization

Text formats (JSON, XML) are prohibited for critical snapshots due to latency and ambiguity. The configuration allows `snapshot_format` values `msgpack` or `protobuf`, but the current IO implementation supports MessagePack only.

- Encode all fields deterministically.
- Preserve float64 for numerical fidelity.

5.2.1 Implementation Notes

The snapshot serializer currently supports MessagePack only (`snapshot_format = "msgpack"`); protobuf is reserved for a future implementation. Hash verification is performed before state injection.

5.3 Integrity Verification

Each snapshot Σ_t must include a hash footer (SHA-256 or CRC32c). The load routine must verify the hash before injecting state into memory.

- Fail closed if hash verification fails.
- Log integrity failures at critical severity.

5.4 Atomic Write Protocol

To avoid partial writes, persist snapshots to a temporary file and then atomically rename to the target path. The rename step must be the only visible operation to consumers.

- Use a unique temporary filename per snapshot.
- Ensure the target file is replaced atomically.

5.4.1 Implementation Notes

Snapshots are written to a unique temporary file and moved into place using atomic rename. Optional fsync ensures persistence across power loss.

Chapter 6

Security Policies

6.1 Credential Injection

Tokens and API keys must not appear in source code. Credentials must be injected at runtime via environment variables or .env files.

6.1.1 Implementation Notes

Credential helpers read from environment variables or .env files and raise explicit errors on missing values.

6.2 Version Control Exclusion

The repository must exclude .env files and credential directories via .gitignore. Secrets must never be committed.

Chapter 7

Orchestrator Integration

7.1 Ingestion Gate in orchestrate_step()

The core orchestration pipeline now integrates ingestion validation as a pre-kernel gate. The `orchestrate_step()` function signature is extended to accept observation metadata:

```
def orchestrate_step(
    signal: Float[Array, "n"],
    timestamp_ns: int,
    state: InternalState,
    config: PredictorConfig,
    observation: ProcessState,
    now_ns: int,
    telemetry_buffer: Optional[TelemetryBuffer] = None,
    step_counter: int = 0,
    mutation_rate_limiter: Optional[MutationRateLimiter] = None,
    degradation_monitor: Optional[DegradationMonitor] = None,
    allow_host_scaling: bool = True,
) -> OrchestrationResult:
    """Run a single orchestration step with IO ingestion validation."""
```

Note: `OrchestrationResult` now carries an optional `config` field to propagate host-side DGM scaling decisions across steps.

7.1.1 Execution Flow

The ingestion gate operates as follows:

1. **Input Validation**: Standard signal length and dtype checks.
2. **Ingestion Decision**: Call `evaluate_ingestion()` with current state, observation, and configuration.
3. **Rejection Logic**: If `accept_observation == False`, reject the entire observation without state update (emergency mode).
4. **Degradation Flags**: Apply `suspend_jko_update` and `freeze_kernel_d` flags to control fusion behavior.
5. **Kernel Execution**: Run kernels A-D; if `freeze_kernel_d == True`, mark kernel D output as frozen.
6. **Host Scaling (optional)**: If `allow_host_scaling == True`, the host may re-run Kernel B with a scaled DGM architecture and return the updated config for the next step.
7. **Fusion Selection**: Skip JKO/Sinkhorn if degraded mode or `suspend_jko_update` is set.

8. **State Update**: Only update InternalState if observation is accepted.

7.1.2 Flag Semantics

The `IngestionDecision` object carries the following flags:

- **accept_observation**: If False, reject and preserve inertial state.
- **suspend_jko_update**: If True, freeze weights and skip Sinkhorn.
- **degraded_mode**: If True, emit degraded inference mode prediction.
- **freeze_kernel_d**: If True, mark kernel D output as frozen (no weight update).
- **staleness_ns**: Staleness in nanoseconds for audit logging.
- **events**: Emitted validation events (outliers, frozen signals, staleness alarms).

7.1.3 Early Return on Rejection

If an observation is rejected (catastrophic outlier), the orchestrator returns a degraded result without advancing the state:

```
# If observation is rejected, skip state update entirely
if reject_observation:
    updated_state = state
else:
    updated_state = atomic_state_update(...)
```

7.2 PRNG Configuration

To eliminate magic numbers in PRNG splitting, the split count is configuration-driven via `core.prng_split_count`

```
[core]
prng_seed = 42
prng_split_count = 4
```

The orchestrator uses `config.prng_split_count` when advancing the RNG key. All PRNG parameters reside in configuration.

7.3 64-bit Precision Enforcement

To ensure Malliavin calculus and Signature computation stability, 64-bit precision must be activated at module import time, before any XLA tracing:

```
# api/config.py: JAX CONFIGURATION (at module level)
import jax
jax.config.update("jax_enable_x64", True)
```

This enforces bit-exact reproducibility across CPU/GPU/FPGA backends and must execute before `ConfigManager` initialization.

Chapter 8

Telemetry Buffer Integration (P2.3)

8.1 Motivation

Phase 2 implementations (P2.1 WTMM, P2.2 SDE stiffness, V-MAJ violations) generate rich diagnostic data during orchestration. To enable post-mortem analysis, compliance audits, and debugging without stalling inference, P2.3 integrates a **non-blocking telemetry buffer** into the orchestration pipeline.

Key requirements:

- **Non-Blocking**: Logging never blocks compute threads (async enqueue only)
- **Thread-Safe**: Multiple consumers can safely drain buffer
- **Audit Trail**: Records capture complete prediction state snapshot
- **Integrity**: Parity hashes verify weights and free energy
- **Config-Driven**: Emission interval and buffer capacity injected from config

8.2 Data Model

Each telemetry record captures:

```python
@dataclass(frozen=True)
class TelemetryRecord:
    step: int                          # Monotonic counter
    payload: dict                      # Rich diagnostic data


# Payload structure (P2.3):
payload = {
    "step": 42,
    "timestamp_ns": 1708308000000000000,  # Nanosecond precision
    "prediction_ref": fused_prediction,   # DeviceArray reference
    "weights_ref": final_rho,             # DeviceArray reference
    "free_energy_ref": free_energy,       # DeviceArray reference
    "kurtosis_ref": updated_state.kurtosis,
    "holder_exponent_ref": updated_state.holder_exponent,
    "dgm_entropy_ref": updated_state.dgm_entropy,
    "mode_collapse_warning": False,       # V-MAJ-5 flag
    "degraded_mode": False,               # V-MAJ-7 hysteresis
    "emergency_mode": False               # Circuit breaker
}
```

Parity hashes are computed during `materialize_telemetry_batch()` after `jax.device_get()` to avoid host-device synchronization in the orchestrator path.

8.3 Integration into orchestrate_step()

The orchestrator now accepts optional `telemetry_buffer` and `step_counter` parameters:

```
def orchestrate_step(
    signal: Float[Array, "n"],
    timestamp_ns: int,
    state: InternalState,
    config: PredictorConfig,
    observation: ProcessState,
    now_ns: int,
    telemetry_buffer: Optional[TelemetryBuffer] = None,  # P2.3
    step_counter: int = 0,  # P2.3
    mutation_rate_limiter: Optional[MutationRateLimiter] = None,
    degradation_monitor: Optional[DegradationMonitor] = None,
) -> OrchestrationResult:
    """
    Run a single orchestration step with telemetry buffering.

    If telemetry_buffer is None, skips telemetry (backward compatible).
    If provided, enqueues record when hash_interval triggers.
    """
    # ... existing orchestration logic ...

    # P2.3: Telemetry Buffer Integration (before return)
    if telemetry_buffer is not None:
        telemetry_payload = {
            "step": step_counter,
            "timestamp_ns": timestamp_ns,
            "prediction_ref": fused_prediction,
            "weights_ref": final_rho,
            "free_energy_ref": free_energy if fusion is not None else jnp.array(0.0),
            "kurtosis_ref": updated_state.kurtosis,
            "holder_exponent_ref": updated_state.holder_exponent,
            "dgm_entropy_ref": updated_state.dgm_entropy,
            "mode_collapse_warning": mode_collapse_warning,
            "degraded_mode": degraded_mode,
            "emergency_mode": emergency_mode,
        }

        # Emit only when hash_interval triggers (config-driven)
        if should_emit_hash(step_counter, config.telemetry_hash_interval_steps):
            telemetry_record = TelemetryRecord(step=step_counter, payload=
    telemetry_payload)
            telemetry_buffer.enqueue(telemetry_record)

    return OrchestrationResult(...)
```

8.4 Configuration Parameters

Parameter	Default	Purpose
`telemetry_buffer_capacity`	1024	Maximum records in ring buffer
`telemetry_hash_interval_steps`	1	Emit telemetry every N steps

Table 8.1: P2.3 Telemetry Configuration

8.4.1 Configuration Injection Example

```toml
# config.toml
[io]
# Telemetry
telemetry_hash_interval_steps = 1          # Emit every step (or 10 for sparser logging)
telemetry_buffer_capacity = 1024           # Ring buffer size (zero-heuristics injection)
```

8.5 Thread-Safety Model

`TelemetryBuffer` uses `threading.Lock` for atomic operations:

- **enqueue()**: Acquires lock, appends record, releases (O(1) amortized)
- **drain()**: Acquires lock, extracts all records, clears buffer, releases
- **size()**: Acquires lock, returns current count, releases

This prevents race conditions when orchestrator (compute thread) enqueues while consumer thread drains.

8.6 Backward Compatibility

P2.3 is fully backward compatible:

- If `telemetry_buffer=None`, no telemetry is emitted (default)
- Existing calls to `orchestrate_step()` without telemetry params continue working
- No changes to compute path (telemetry entirely outside @jax.jit scope)

8.7 Usage Example

```python
from Python.io.telemetry import TelemetryBuffer
from Python.api.config import PredictorConfigInjector

# Initialize
config = PredictorConfigInjector().create_config()
telemetry_buffer = TelemetryBuffer(capacity=config.telemetry_buffer_capacity)

# In prediction loop:
for step in range(num_steps):
    result = orchestrate_step(
        signal=current_signal,
        timestamp_ns=now_ns(),
        state=state,
        config=config,
        observation=obs,
        now_ns=now_ns(),
        telemetry_buffer=telemetry_buffer,     # P2.3: Pass buffer
        step_counter=step,                     # P2.3: Pass step
    )

    # Optional: Drain telemetry in background thread
    if step\% 100 == 0:
        records = telemetry_buffer.drain()
        # Write to file/database (non-blocking, doesn't stall orchestrator)
        write_telemetry_async(records)
```

8.8 Benefits

- **Non-Blocking**: Telemetry enqueue is O(1), never stalls inference
- **Audit Trail**: Complete state snapshots for compliance and debugging
- **Integrity**: Parity hashes enable CPU/GPU parity validation
- **Configurable**: Emission interval and buffer size injected from config
- **Thread-Safe**: Lock-based synchronization for multi-threaded consumers
- **Backward Compatible**: Fully optional, no impact on existing code paths

8.9 Visualization Dashboard (GAP-6)

The IO layer includes a static HTML dashboard generator for telemetry snapshots. It requires no external dependencies and produces a standalone report suitable for offline reviews and audits.

8.9.1 Usage

```
from Python.io.telemetry import TelemetryBuffer
from Python.io.dashboard import export_dashboard_snapshot

telemetry_buffer = TelemetryBuffer(capacity=1024)
# ... enqueue telemetry records during inference ...

exported = export_dashboard_snapshot(
    telemetry_buffer,
    "io/reports/telemetry_dashboard.html",
)
print(f"Exported {exported} records")
```

8.9.2 Notes

- The dashboard drains the buffer, materializes device arrays, and writes HTML.
- Charts are rendered as inline SVG with no external assets.
- Output is deterministic and suitable for audit trails.

Chapter 9

Level 4 Autonomy: Configuration Mutation Safety

9.1 Overview

Phase 2.1.0 introduces **Level 4 Autonomy** compliance for autonomous configuration mutations during meta-optimization. This chapter documents the implementation of V-CRIT-2, V-MAJ-4, V-MAJ-5, and V-MAJ-7 violations identified during the specification compliance audit (AUDIT_SPEC_COMPLIANCE_2026-02-19.md).

Specification References:

- `Stochastic_Predictor_IO.tex` §3.3 - Configuration Mutation Protocol (Atomic Write)
- `Stochastic_Predictor_IO.tex` §3.3.6 - Rate Limiting and Safety Guardrails
- `Stochastic_Predictor_Implementation.tex` §5.4.3 - Degradation Detection Protocol
- `Stochastic_Predictor_Theory.tex` §2.3.6 - Monitoring and Telemetry for adaptive SDE schemes

Implementation Scope:

- **V-CRIT-2**: Atomic TOML mutation protocol with locked subsection protection
- V-MAJ-4: Configuration mutation rate limiting
- V-MAJ-5: Degradation detection with automatic rollback
- V-MAJ-7: Adaptive telemetry monitoring (partial - safety guardrails)

9.2 V-CRIT-2: Atomic Configuration Mutation Protocol

9.2.1 Problem Statement

Violation: No atomic write mechanism for config.toml mutations during autonomous meta-optimization. Race conditions, partial writes, and locked parameter violations could corrupt system configuration without rollback capability.

Impact:

- Partial file corruption during concurrent writes
- No protection for immutable subsections (e.g., float_precision, snapshot_format)
- No audit trail for forensic analysis
- Cannot rollback to previous working configuration
- Loss of checkpoint resumability if meta-optimization paths corrupted

9.2.2 Requirements

IO.tex §3.3.2 - Atomic TOML Update Algorithm:

1. **Phase 1 - Validation**: Check parameter ranges, types, locked subsections
2. **Phase 2 - Backup**: Create timestamped backup + latest .bak
3. **Phase 3 - Atomic Write**: Write to .tmp, fsync(), os.replace()
4. **Phase 4 - Audit Log**: Record mutation to io/mutations.log
5. **Phase 5 - Success**: Return or raise on failure

IO.tex §3.3.4 - Locked Subsections (Asimov's Zeroth Law):

- `[meta]`: schema_version
- `[core]`: jax_platforms, jax_default_dtype, float_precision, staleness_ttl_ns
- `[io]`: snapshot_format, snapshot_hash_algorithm, snapshot_compression, telemetry_hash_interval_steps
- `[meta_optimization]`: n_trials, n_startup_trials, multivariate, train_ratio, n_folds

9.2.3 Implementation

Module: `stochastic_predictor/io/config_mutation.py` (EXTENDED)
Functions Added:

- `atomic_write_config()`: 5-phase POSIX-compliant atomic mutation
- `validate_config_mutation()`: Schema validation + locked subsection checks
- `append_audit_log()`: JSON Lines audit trail logging
- `create_config_backup()`: Timestamped backup creation

Constants Added:

- `LOCKED_SUBSECTIONS`: Immutable parameter dictionary
- `VALIDATION_SCHEMA`: Type/range constraints for mutable parameters

Atomic Write Protocol Implementation

```
def atomic_write_config(
    config_path: Path,
    new_params: Dict[str, Any],
    trigger: str = "ManualMutation",
    best_objective: Optional[float] = None,
    audit_log_path: Optional[Path] = None,
) -> None:
    """
    Atomically mutate configuration with POSIX-compliant write protocol.

    COMPLIANCE: IO.tex §3.3.2 - Atomic TOML Update Algorithm
    """
    audit_log_path = audit_log_path or Path("io/mutations.log")

    # Phase 1: Validation
    current_config = toml.load(config_path)
```

```python
    merged_config = validate_config_mutation(current_config, new_params)

    # Compute delta for audit log
    delta = {
        param_key: (_get_nested_param(current_config, param_key), new_value)
        for param_key, new_value in new_params.items()
    }

    # Phase 2: Immutable Backup
    timestamp = datetime.now(timezone.utc).strftime("%Y-%m-%dT%H:%M:%SZ")
    backup_path = config_path.with_suffix(f".bak.{timestamp}")
    latest_backup_path = config_path.with_suffix(".bak")

    shutil.copy2(config_path, backup_path)
    shutil.copy2(config_path, latest_backup_path)

    # Phase 3: Atomic Write via Temporary File
    tmp_path = config_path.with_suffix(".tmp")

    # O_EXCL flag detects concurrent mutation
    fd = os.open(tmp_path, os.O_WRONLY | os.O_CREAT | os.O_EXCL, 0o644)

    try:
        toml_bytes = toml.dumps(merged_config).encode("utf-8")
        os.write(fd, toml_bytes)

        # CRITICAL: fsync() ensures kernel buffer flush to disk
        os.fsync(fd)
    finally:
        os.close(fd)

    # Phase 4: Atomic Replacement (POSIX os.replace)
    os.replace(tmp_path, config_path)

    # Phase 5: Audit Logging
    append_audit_log(audit_log_path, {
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "event": "MUTATION_SUCCESS",
        "trigger": trigger,
        "delta": delta,
        "best_objective": best_objective,
        "backup": str(backup_path),
        "status": "SUCCESS",
    })
```

Validation Schema Implementation

```python
# Locked subsections (immutable to prevent self-corruption)
LOCKED_SUBSECTIONS = {
    "meta": ["schema_version"],
    "core": ["jax_platforms", "jax_default_dtype", "float_precision", "staleness_ttl_ns"
    ],
    "io": [
        "snapshot_format",
        "snapshot_hash_algorithm",
        "snapshot_compression",
        "telemetry_hash_interval_steps",
    ],
    "meta_optimization": ["n_trials", "n_startup_trials", "multivariate", "train_ratio",
    "n_folds"],
}

```

```python
# Validation schema for mutable parameters
VALIDATION_SCHEMA = {
    "orchestration.cusum_k": {"type": float, "range": (0.1, 1.0)},
    "orchestration.cusum_h": {"type": float, "range": (2.0, 10.0)},
    "orchestration.grace_period_steps": {"type": int, "range": (5, 100)},
    "orchestration.volatility_alpha": {"type": float, "range": (0.05, 0.3)},
    "orchestration.learning_rate": {"type": float, "range": (1e-5, 1e-1)},
    "orchestration.entropy_window": {"type": int, "range": (10, 500)},
    "orchestration.entropy_threshold": {"type": float, "range": (0.5, 0.95)},
    "orchestration.holder_threshold": {"type": float, "range": (0.2, 0.65)},
    "orchestration.sinkhorn_alpha": {"type": float, "range": (0.1, 1.0)},
    "orchestration.sinkhorn_epsilon_min": {"type": float, "range": (0.001, 0.1)},
    "orchestration.sinkhorn_epsilon_0": {"type": float, "range": (0.05, 0.5)},
    "orchestration.sinkhorn_max_iter": {"type": int, "range": (50, 500)},
    "kernels.log_sig_depth": {"type": int, "range": (2, 5)},
    "kernels.wtmm_buffer_size": {"type": int, "range": (64, 512),
                                 "constraint": "power_of_2"},
    "kernels.besov_cone_c": {"type": float, "range": (1.0, 3.0)},
    "kernels.dgm_width_size": {"type": int, "range": (32, 256),
                                 "constraint": "power_of_2"},
    "kernels.stiffness_low": {"type": float, "range": (50.0, 500.0)},
    "kernels.stiffness_high": {"type": float, "range": (500.0, 5000.0)},
    "kernels.sde_dt": {"type": float, "range": (0.001, 0.1)},
    "kernels.sde_numel_integrations": {"type": int, "range": (50, 200)},
    "kernels.sde_diffusion_sigma": {"type": float, "range": (0.05, 0.5)},
    "kernels.kernel_ridge_lambda": {"type": float, "range": (1e-8, 1e-3)},
    # ... additional mutable parameters
}

def validate_config_mutation(
    current_config: Dict[str, Any],
    new_params: Dict[str, Any],
) -> Dict[str, Any]:
    """
    Validate configuration mutation against schema and locked subsections.

    Raises:
        ConfigMutationError: If locked parameter mutation attempted
        ConfigMutationError: If parameter out of safe range
    """
    # Check locked subsection violations
    for param_key in new_params.keys():
        subsection = param_key.split(".")[0]
        param_name = ".".join(param_key.split(".")[1:])

        if subsection in LOCKED_SUBSECTIONS:
            if param_name in LOCKED_SUBSECTIONS[subsection]:
                raise ConfigMutationError(
                    f"Parameter '{param_key}' is LOCKED (immutable)"
                )

    # Validate against schema (ranges, types, constraints)
    merged_config = dict(current_config)
    for param_key, new_value in new_params.items():
        rules = VALIDATION_SCHEMA[param_key]

        # Type check
        if not isinstance(new_value, rules["type"]):
            raise ConfigMutationError("Type mismatch")

        # Range check
        min_val, max_val = rules["range"]
        if not (min_val <= new_value <= max_val):
```

```python
            raise ConfigMutationError("Out of safe range")

        # Apply constraints (e.g., power_of_2)
        if "constraint" in rules:
            # Check constraint...

        _set_nested_param(merged_config, param_key, new_value)

    # Cross-parameter constraints (e.g., stiffness_low < stiffness_high)
    # ...

    return merged_config
```

9.2.4 Usage Example

```python
from Python.io import atomic_write_config

# After Deep Tuning completes
best_params = {
    "orchestration.cusum_k": 0.72,
    "kernels.dgm_width_size": 256,
    "kernels.stiffness_low": 143.0,
}

# Atomic mutation with audit trail
atomic_write_config(
    Path("config.toml"),
    best_params,
    trigger="DeepTuning_Iteration_500",
    best_objective=0.0234  # MAPE
)

# Creates:
#   - config.toml.bak.2026-02-19T14:32:05Z (timestamped backup)
#   - config.toml.bak (latest backup)
#   - io/mutations.log (audit entry)
```

9.2.5 Files Modified

- `stochastic_predictor/io/config_mutation.py`: +280 LOC
- `stochastic_predictor/io/__init__.py`: +7 exports

9.2.6 Compliance Impact

V-CRIT-2 Resolution: Atomic TOML mutation protocol fully implemented with:

- POSIX-compliant atomic write (fsync + os.replace)
- Locked subsection protection (Asimov's Zeroth Law)
- Validation schema enforcement (20+ mutable parameters)
- Timestamped backups for manual rollback
- JSON Lines audit trail for forensic analysis

Level 4 Autonomy Enablement: System can now autonomously mutate config.toml during Deep Tuning campaigns without human intervention, while maintaining safety guarantees.

9.3 V-MAJ-4: Configuration Mutation Rate Limiting

9.3.1 Problem Statement

Violation: No safety guardrails enforced for autonomous configuration mutations during meta-optimization. Optimizer could thrash between configurations, mutate too frequently, or make excessively large parameter jumps.

Impact: System instability, pathological optimizer behavior, configuration thrashing without convergence.

9.3.2 Safety Requirements

IO.tex §3.3.6 Requirements:

- Maximum mutation rate: 10 mutations/hour
- Minimum stability period: 1,000 prediction steps between mutations
- Delta magnitude limit: 50% relative change per parameter
- Audit trail: JSON Lines log of all mutation events

9.3.3 Implementation

Module: `stochastic_predictor/io/config_mutation.py` (NEW)

Class: `MutationRateLimiter`

```
@dataclass
class MutationRateLimiter:
    """Enforce safety guardrails for autonomous configuration mutations.

    Prevents optimizer pathologies:
        - Thrashing between configurations
        - Excessive mutation frequency
        - Large parameter jumps
        - Pathological degradation without rollback

    Args:
        max_mutations_per_hour: Maximum allowed mutations/hour (default 10)
        stability_steps_required: Minimum steps before next mutation (default 1000)
        max_relative_change: Maximum parameter change per mutation (default 0.5)
    """

    max_mutations_per_hour: int = 10
    stability_steps_required: int = 1000
    max_relative_change: float = 0.5

    _mutation_history: List[Tuple[float, Dict]] = field(default_factory=list)
    _last_mutation_timestamp: Optional[float] = None
    _current_steps_since_mutation: int = 0

    def can_mutate(self) -> Tuple[bool, str]:
        """Check if mutation is allowed under safety guardrails.

        Returns:
            (allowed: bool, reason: str)
        """
        # Check maximum mutation rate (sliding 1-hour window)
        one_hour_ago = time.time() - 3600
        recent_mutations = [
            ts for ts, _ in self._mutation_history if ts > one_hour_ago
```

```python
        ]
        if len(recent_mutations) >= self.max_mutations_per_hour:
            return False, f"Rate limit: {len(recent_mutations)}/10 mutations"

        # Check minimum stability period
        if self._current_steps_since_mutation < self.stability_steps_required:
            return False, f"Stability: {self._current_steps_since_mutation}/1000"

        return True, "OK"

    def validate_delta(
        self,
        delta: Dict[str, Tuple[float, float]],
        max_relative_change: Optional[float] = None
    ) -> Tuple[bool, str]:
        """Validate parameter delta magnitude.

        Args:
            delta: {param: (old_value, new_value)}
            max_relative_change: Override default max change

        Returns:
            (valid: bool, reason: str)
        """
        max_change = max_relative_change or self.max_relative_change

        for param, (old_val, new_val) in delta.items():
            if abs(old_val) < 1e-12:
                continue  # Skip zero division

            relative_change = abs((new_val - old_val) / old_val)

            if relative_change > max_change:
                return False, (
                    f"{param} change too large: {relative_change:.1%} > 50%"
                )

        return True, "OK"

    def record_mutation(self, delta: Dict) -> None:
        """Record successful mutation."""
        now = time.time()
        self._mutation_history.append((now, delta))
        self._current_steps_since_mutation = 0

    def increment_stability_counter(self) -> None:
        """Call after each prediction step."""
        self._current_steps_since_mutation += 1
```

9.3.4 Usage Pattern

```python
# In meta-optimizer loop
limiter = MutationRateLimiter(max_mutations_per_hour=10)

# Export best params using atomic mutation protocol
optimizer.export_best_params_to_config(
    "config.toml",
    trigger="DeepTuning_Iteration_500",
    rate_limiter=limiter,
)

# In main prediction loop
```

```python
for step in range(1000):
    prediction = orchestrate_step(..., mutation_rate_limiter=limiter)
    # If not passing mutation_rate_limiter, call limiter.increment_stability_counter()
    manually
```

9.4 V-MAJ-5: Degradation Detection Auto-Rollback

9.4.1 Problem Statement

Violation: No automatic rollback mechanism when post-mutation performance degrades beyond acceptable threshold. Pathological mutations persist indefinitely, requiring manual operator intervention.

Impact: System can enter degraded state without automatic recovery, violating Level 4 autonomy requirements.

9.4.2 Safety Requirements

IO.tex §3.3.6 Requirements:

- Monitor RMSE over N=100 predictions post-mutation
- Compare to pre-mutation baseline RMSE
- If relative increase > 30%, trigger automatic rollback
- Restore `config.toml` from `config.toml.bak`
- Log rollback event to audit trail

9.4.3 Implementation

Module: `stochastic_predictor/io/config_mutation.py`

Class: `DegradationMonitor`

```python
@dataclass
class DegradationMonitor:
    """Monitor post-mutation performance and trigger rollback on degradation.

    Implements closed-loop safety mechanism:
        1. Record pre-mutation baseline RMSE
        2. Monitor post-mutation predictions (N=100 sample window)
        3. Compute post-mutation RMSE
        4. If relative increase > threshold (default 30%), auto-rollback

    Args:
        degradation_threshold: Max allowed RMSE increase (default 0.3 = 30%)
        monitoring_window: Predictions to sample post-mutation (default 100)
        config_path: Path to config.toml (default "config.toml")
        backup_path: Path to backup config (default "config.toml.bak")
        audit_log_path: Path to mutation audit log
    """

    degradation_threshold: float = 0.3
    monitoring_window: int = 100
    config_path: Path = Path("config.toml")
    backup_path: Path = Path("config.toml.bak")
    audit_log_path: Path = Path("io/mutations.log")

    _pre_mutation_rmse: Optional[float] = None
```

```python
    _post_mutation_errors: List[float] = field(default_factory=list)
    _monitoring_active: bool = False

    def start_monitoring(self, baseline_rmse: float) -> None:
        """Record pre-mutation baseline and start monitoring."""
        self._pre_mutation_rmse = baseline_rmse
        self._post_mutation_errors = []
        self._monitoring_active = True

    def record_prediction_error(self, error: float) -> None:
        """Accumulate post-mutation prediction error."""
        if not self._monitoring_active:
            return
        self._post_mutation_errors.append(error)

    def check_degradation(self) -> Tuple[bool, float]:
        """Check if post-mutation performance degraded beyond threshold.

        Returns:
            (degraded: bool, relative_increase: float)
        """
        if not self._monitoring_active:
            return False, 0.0

        # Require full monitoring window
        if len(self._post_mutation_errors) < self.monitoring_window:
            return False, 0.0

        # Compute post-mutation RMSE
        post_mutation_rmse = np.sqrt(
            np.mean(np.square(self._post_mutation_errors))
        )

        # Compute relative increase
        if self._pre_mutation_rmse is None or self._pre_mutation_rmse == 0:
            relative_increase = 0.0
        else:
            relative_increase = (
                (post_mutation_rmse - self._pre_mutation_rmse) /
                self._pre_mutation_rmse
            )

        degraded = relative_increase > self.degradation_threshold

        # Stop monitoring after check
        if degraded or len(self._post_mutation_errors) >= self.monitoring_window:
            self._monitoring_active = False

        return degraded, relative_increase

    def trigger_rollback(self) -> None:
        """Execute automatic rollback to pre-mutation configuration.

        CRITICAL: Overwrites config.toml with backup.
        """
        if not self.backup_path.exists():
            raise FileNotFoundError(f"Backup config not found: {self.backup_path}")

        # Restore config from backup
        shutil.copy2(self.backup_path, self.config_path)

        # Compute post-mutation RMSE for logging
        post_mutation_rmse = float(np.sqrt(
```

```
            np.mean(np.square(self._post_mutation_errors))
        )) if self._post_mutation_errors else 0.0

        # Append rollback event to audit log (JSON Lines format)
        audit_entry = {
            'timestamp': datetime.now(timezone.utc).isoformat(),
            'event': 'AUTO_ROLLBACK',
            'reason': 'Performance degradation detected',
            'pre_mutation_rmse': self._pre_mutation_rmse,
            'post_mutation_rmse': post_mutation_rmse,
            'relative_increase': (
                (post_mutation_rmse - self._pre_mutation_rmse) /
                self._pre_mutation_rmse
            ),
            'degradation_threshold': self.degradation_threshold,
            'monitoring_window': self.monitoring_window,
            'status': 'ROLLBACK_SUCCESS'
        }

        # Append to audit log
        self.audit_log_path.parent.mkdir(parents=True, exist_ok=True)
        with open(self.audit_log_path, 'a') as f:
            f.write(json.dumps(audit_entry) + '\n')

        # Reset monitoring state
        self._monitoring_active = False
        self._post_mutation_errors = []
```

9.4.4 Usage Pattern

```
# Before mutation
monitor = DegradationMonitor(degradation_threshold=0.3)

# Compute baseline RMSE over recent predictions
recent_errors = [0.04, 0.05, 0.06, 0.05, 0.04]
baseline_rmse = np.sqrt(np.mean(np.square(recent_errors)))

# Apply mutation
delta = {"orchestration.learning_rate": 0.015}
atomic_write_config(Path("config.toml"), delta, trigger="AutoTuning")

# Start monitoring
monitor.start_monitoring(baseline_rmse)

# In prediction loop (next 100 predictions)
for i in range(100):
    prediction = orchestrate_step(..., degradation_monitor=monitor)
    actual_next = load_actual_observation()
    error = abs(prediction.predicted_next - actual_next)

    monitor.record_prediction_error(error)

    # Check for degradation
    degraded, increase = monitor.check_degradation()
    if degraded:
        logger.critical(f"Degradation detected: RMSE +{increase:.1%}")
        monitor.trigger_rollback()
        logger.info("Config rolled back to pre-mutation state")
        break
```

9.4.5 Audit Log Format

Mutation events are logged in JSON Lines format to `io/mutations.log`:


```
{"timestamp": "2026-02-19T14:30:00Z", "event": "MUTATION_SUCCESS",
 "delta": {"orchestration.learning_rate": [0.01, 0.015]}, "status": "SUCCESS"}
{"timestamp": "2026-02-19T14:35:00Z", "event": "AUTO_ROLLBACK",
 "reason": "Performance degradation detected",
 "pre_mutation_rmse": 0.05, "post_mutation_rmse": 0.07,
 "relative_increase": 0.40, "degradation_threshold": 0.30,
 "status": "ROLLBACK_SUCCESS"}
```


9.5 V-MAJ-7: Adaptive Telemetry Monitoring

9.5.1 Implementation Status

V-MAJ-7 requires comprehensive telemetry for adaptive architecture and solver selection metrics. This implementation extends `InternalState` with counters for Level 4 autonomy monitoring and provides full adaptive telemetry collection.

9.5.2 InternalState Extensions

Module: `stochastic_predictor/api/types.py`

Added fields to `InternalState` (lines 443-447):

```
# V-MAJ-7: Level 4 Autonomy Adaptive Telemetry
baseline_entropy: Float[Array, "1"]     # H_baseline: Reference entropy for
solver_explicit_count: int              # N_explicit: Explicit SDE solver steps
solver_implicit_count: int              # N_implicit: Implicit SDE solver steps
architecture_scaling_events: int        # N_scale: DGM architecture scaling events
```

Purpose:

- `baseline_entropy`: Reference entropy H_{baseline} for computing entropy ratio $\kappa = H_{\text{current}}/H_{\text{baseline}}$
- `solver_explicit_count`, `solver_implicit_count`: Track SDE solver scheme frequencies within monitoring window (default 100 steps)
- `architecture_scaling_events`: Count DGM architecture scaling events (cumulative)

9.5.3 Adaptive Telemetry Collection

Module: `stochastic_predictor/io/telemetry.py`

Function: `collect_adaptive_telemetry()`

```
def collect_adaptive_telemetry(
    state: Any,  # InternalState
    config: Any,  # PredictorConfig
    window_size: int = 100
) -> Optional[AdaptiveTelemetry]:
    """
    Collect telemetry for adaptive architecture/solver diagnostics.

    COMPLIANCE: V-MAJ-7 - Adaptive Telemetry Monitoring

    Args:
        state: Current InternalState (contains counters, entropy, etc.)
        config: Current PredictorConfig (may have been mutated)
        window_size: Monitoring window for frequency calculations

```

```
    Returns:
        AdaptiveTelemetry instance or None if insufficient data
    """
    import jax.numpy as jnp

    # Compute solver frequencies (clipped to [0,1])
    total_solver_steps = state.solver_explicit_count + state.solver_implicit_count
    if total_solver_steps == 0:
        return None  # Insufficient data

    freq_explicit = float(state.solver_explicit_count) / total_solver_steps
    freq_implicit = float(state.solver_implicit_count) / total_solver_steps

    # Compute entropy ratio   = H_current / H_baseline
    baseline_entropy_val = float(state.baseline_entropy)
    if baseline_entropy_val <= 0.0:
        entropy_ratio = 1.0  # Avoid division by zero
    else:
        entropy_ratio = float(state.dgm_entropy) / baseline_entropy_val

    # Extract DGM architecture from config
    dgm_width = config.dgm_width_size
    dgm_depth = config.dgm_depth

    # Extract JKO flow parameters from config
    entropy_window = config.entropy_window
    learning_rate = config.learning_rate
    volatility_sigma_squared = float(state.ema_variance)

    # Extract adaptive stiffness thresholds from config
    stiffness_low = config.stiffness_low
    stiffness_high = config.stiffness_high
    holder_exponent_val = float(state.holder_exponent)

    return AdaptiveTelemetry(
        # SDE Solver Frequency
        scheme_frequency_explicit=freq_explicit,
        scheme_frequency_implicit=freq_implicit,
        max_stiffness_metric=0.0,  # Placeholder - requires Kernel C integration
        num_internal_iterations_mean=0.0,  # Placeholder
        implicit_residual_norm_max=0.0,  # Placeholder

        # DGM Architecture
        entropy_ratio_current=entropy_ratio,
        dgm_width_current=dgm_width,
        dgm_depth_current=dgm_depth,
        architecture_scaling_events=state.architecture_scaling_events,

        # JKO Flow
        entropy_window_current=entropy_window,
        learning_rate_current=learning_rate,
        volatility_sigma_squared=volatility_sigma_squared,

        # Stiffness Thresholds
        stiffness_low_adaptive=stiffness_low,
        stiffness_high_adaptive=stiffness_high,
        holder_exponent_wtmm=holder_exponent_val
    )
```

9.5.4 Integration Pattern

Adaptive telemetry collection is called periodically (e.g., every 100 steps) in the orchestration loop:

```python
# In orchestrate_step() or meta-loop
if step % telemetry_interval == 0:
    adaptive_tel = collect_adaptive_telemetry(state, config)
    if adaptive_tel:
        emit_adaptive_telemetry(adaptive_tel)
```

Output: JSON Lines format appended to `io/adaptive_telemetry.jsonl`

9.5.5 Metrics Tracked

SDE Solver Frequency:

- `scheme_frequency_explicit`: Fraction of explicit Euler steps in window $\in [0, 1]$
- `scheme_frequency_implicit`: Fraction of implicit solver steps in window $\in [0, 1]$

DGM Architecture:

- `entropy_ratio_current`: $\kappa = H_{\text{current}}/H_{\text{baseline}}$ (regime transition detector)
- `dgm_width_current`, `dgm_depth_current`: Current DGM network dimensions (may be scaled dynamically)
- `architecture_scaling_events`: Cumulative count of architecture mutations

JKO Flow:

- `entropy_window_current`: Current entropy window $W \propto L^2/\sigma^2$
- `learning_rate_current`: Current JKO learning rate $\eta < 2\varepsilon\sigma^2$
- `volatility_sigma_squared`: Current EWMA variance σ^2

Stiffness Thresholds:

- `stiffness_low_adaptive`, `stiffness_high_adaptive`: Hölder-informed thresholds $\theta_L, \theta_H \propto 1/(1-\alpha)^2$
- `holder_exponent_wtmm`: Current Hölder exponent $\alpha \in [0.2, 0.9]$

9.5.6 Future Extensions

- `max_stiffness_metric`: Maximum stiffness metric $\mathcal{S}$ in window (requires Kernel C integration)
- `num_internal_iterations_mean`: Average Newton iterations for implicit solver
- `implicit_residual_norm_max`: Maximum residual norm for implicit solver convergence

These metrics require deeper integration with Kernel C's SDE solver internals and are currently placeholders (set to 0.0).

9.6 Utility Functions

Module: `stochastic_predictor/io/config_mutation.py`

```python
def create_config_backup(
    config_path: Path = Path("config.toml"),
    backup_path: Path = Path("config.toml.bak")
) -> None:
    """Create config backup before mutation."""
    if not config_path.exists():
        raise FileNotFoundError(f"Config file not found: {config_path}")
    shutil.copy2(config_path, backup_path)

def append_audit_log(log_path: Path, entry: Dict) -> None:
    """Append mutation event to audit log (JSON Lines format)."""
    log_path.parent.mkdir(parents=True, exist_ok=True)
    with open(log_path, 'a') as f:
        f.write(json.dumps(entry) + '\n')
```

9.7 Implementation Status

V-MAJ Violation	Status	Module
V-MAJ-4 (Rate Limiting)	Implemented	config_mutation.py
V-MAJ-5 (Auto-Rollback)	Implemented	config_mutation.py
V-MAJ-7 (Telemetry)	Implemented	telemetry.py, types.py

Table 9.1: Level 4 Autonomy - Configuration Mutation Safety Implementation

Note: V-MAJ-7 is fully implemented with `InternalState` extensions (solver counters, baseline entropy, architecture scaling events) and `collect_adaptive_telemetry()` function. Three Kernel C solver metrics remain as placeholders (max_stiffness_metric, num_internal_iterations_mean, implicit_residual_norm_max) requiring deeper SDE solver integration in a future phase.

Chapter 10

Compliance Checklist

- **No Compute Stalls**: All logging is asynchronous
- **Binary Format**: MessagePack for snapshots
- **Atomic Snapshots**: Write-then-rename protocol
- **Deterministic Hashing**: SHA-256 on ρ and OT cost
- **Security**: No raw signals, VRAM dumps, or secrets
- **Integrity**: Snapshot hashes verified before load
- **Config-Driven**: Intervals and destinations are injected
- **Module Coverage**: IO helpers implemented for validation, telemetry, snapshots, and credentials
- **Orchestrator Integration**: IO ingestion gate integrated into `orchestrate_step()`
- **PRNG Constants**: Named constants (RNG_SPLIT_COUNT) reside in `api/prng.py`
- **Buffer Capacity Injection**: TelemetryBuffer capacity injected from config (zero-heuristics policy)
- **64-bit Precision**: Enforced at module load time (`api/config.py`) before XLA tracing
- **Layer Isolation**: PRNG constants in API layer, not Core layer

Chapter 11

Phase 4 Summary

Phase 4 introduces a non-blocking I/O architecture that preserves deterministic compute while enabling telemetry, logging, and atomic snapshot persistence.