Universal Stochastic Predictor
Phase 4: IO Layer Initiation

Implementation Team

February 19, 2026

Contents

Chapter 1

Phase 4: IO Layer Initiation Overview

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.

1.1 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

1.2 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

The frozen signal lock is released after the variance exceeds a configurable ratio of historical variance for a configurable number of consecutive steps.

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.

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 (CUSUM, entropy, regime flags, OT cost).
- 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. Parity hashes are emitted on a configurable interval and derived from canonical float64 serialization.

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. Use dense binary formats such as Protocol Buffers or MessagePack.

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

5.2.1 Implementation Notes

The snapshot serializer uses MessagePack as the default binary format. 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:

```python
def orchestrate_step(
    signal: Float[Array, "n"],
    timestamp_ns: int,
    state: InternalState,
    config: PredictorConfig,
    observation: ProcessState,
    now_ns: int,
) -> OrchestrationResult:
    """Run a single orchestration step with IO ingestion validation."""
```

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. **Fusion Selection**: Skip JKO/Sinkhorn if degraded mode or `suspend_jko_update` is set.
7. **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:

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

7.2 PRNG Constant

To eliminate magic numbers in PRNG splitting, we introduce a module-level constant:

```python
RNG_SPLIT_COUNT = 2
rng_key = jax.random.split(state.rng_key, RNG_SPLIT_COUNT)[1]
```

This ensures clarity and maintainability.

Chapter 8

Compliance Checklist

- **No Compute Stalls**: All logging is asynchronous
- **Binary Format**: Protocol Buffers or 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 replace magic numbers

Chapter 9

Phase 4 Summary

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