

# **Python Test Suite for the Universal Stochastic Predictor**

Adaptive Meta-Prediction Development Consortium

February 21, 2026

# Contents

<b>1 Testing Environment Setup</b>	<b>2</b>
1.1 Dependencies and Tools . . . . .	2
1.2 Directory Structure . . . . .	2
1.3 Shared Fixtures (conftest.py) . . . . .	3
<b>2 Unit Tests: Generation and Analysis</b>	<b>5</b>
2.1 Stable Variable Generation (Chambers-Mallows-Stuck) . . . . .	5
2.2 WTMM Test (Wavelet Transform Modulus Maxima) . . . . .	6
2.3 DGM Entropy Test (Mode Collapse Detection) . . . . .	6
2.4 Property-Based Testing (Hypothesis) . . . . .	7
<b>3 Robustness Tests: CUSUM and Circuit Breakers</b>	<b>10</b>
3.1 Standard CUSUM Test . . . . .	10
3.2 CUSUM with Adaptive Kurtosis . . . . .	11
3.3 Circuit Breaker Test (Singularity) . . . . .	12
<b>4 Integration Tests: DGM and Orchestrator</b>	<b>13</b>
4.1 Deep Galerkin Method Test . . . . .	13
4.2 Sinkhorn and JKO Test . . . . .	14
<b>5 I/O and Persistence Tests</b>	<b>15</b>
5.1 Atomic Snapshotting Test . . . . .	15
<b>6 Hardware Tests: CPU/GPU Parity</b>	<b>17</b>
6.1 Numerical Consistency Test . . . . .	17
6.2 Hardware Parity with Quantization (FPGA Simulation) . . . . .	18
<b>7 XLA VRAM and JIT Cache Assertions</b>	<b>20</b>
7.1 Asynchronous Device Dispatch (No Host-Device Synchronization) . . . . .	20
7.1.1 Telemetry Non-Blocking Guarantee . . . . .	20
7.2 Vectorized Multi-Tenancy (jax.vmap Parity) . . . . .	21
7.2.1 Batch Execution Bit-Exactness . . . . .	21
7.3 JIT Cache Efficiency Under Load Shedding . . . . .	24
7.3.1 Zero-Recompilation Guarantee for Emergency Mode . . . . .	24
7.4 Atomic Configuration Mutation (POSIX Guarantees) . . . . .	25
7.4.1 Temporary File Protocol Enforcement . . . . .	25
<b>8 Edge Cases and Degraded Mode</b>	<b>28</b>
8.1 Degraded Mode Test (TTL Violation) . . . . .	28
8.2 Extreme Kurtosis Test . . . . .	29
<b>9 Walk-Forward Validation</b>	<b>30</b>

<b>10 Strict Causality Validation</b>	<b>32</b>
10.0.1 Causal Mask Test: Intentional Future Poisoning	32
10.0.2 SDE Fuzzing: Extreme Time Steps	32
10.1 No-Clairvoyance via Pointer Inspection	33
<b>11 Test Coverage Summary</b>	<b>36</b>
11.1 Coverage Matrix	36
11.2 Full Suite Execution	36
11.2.1 Environment Validation in CI/CD	36
11.2.2 Execution Commands	37
11.3 Global Acceptance Criteria	37

# Chapter 1

## Testing Environment Setup

### 1.1 Dependencies and Tools

```
1 # requirements-test.txt
2 pytest>=7.4.0
3 pytest-cov>=4.1.0
4 pytest-xdist>=3.3.0 # Test parallelization
5 hypothesis>=6.82.0 # Property-based testing
6 jax[cpu]>=0.4.13 # CPU tests
7 jax[cuda12]>=0.4.13 # GPU tests (optional)
8 numpy>=1.24.0
9 scipy>=1.11.0
10 PyWavelets>=1.4.1
11 msgpack>=1.0.5
12 optuna>=3.2.0
13
14 # Validation tools
15 flake8>=6.0.0
16 mypy>=1.4.0
17 black>=23.7.0
```

### 1.2 Directory Structure

```
1 tests/
2   __init__.py
3   conftest.py           # Shared fixtures
4   test_unit/
5     test_cms_levy.py    # Stable variable generation
6     test_wtmm.py        # Multifractal analysis
7     test_malliavin.py   # Sensitivity computations
8     test_signatures.py  # Branch D
9     test_entropy.py     # DGM entropy
10    test_integration/
11      test_sde_solvers.py # Euler-Maruyama, Milstein
12      test_sinkhorn.py   # Optimal transport
13      test_dgm.py        # Deep Galerkin Method
14      test_orchestrator.py # Full JKO
15    test_robustness/
16      test_cusum.py      # Change detection
17      test_cusum_kurtosis.py # CUSUM with kurtosis
18      test_circuit_breaker.py # Singularities
19      test_outliers.py   # Extreme values
20    test_io/
21      test_snapshotting.py # Persistence
22      test_recovery.py   # Atomic recovery
```

```

23 test_hardware/
24     test_cpu_gpu_parity.py      # Numerical consistency
25     test_numerical_drift.py    # Fixed-point drift
26 test_validation/
27     test_walk_forward.py       # Causal validation
28     test_optuna_tuning.py     # Meta-optimization
29 test_edge_cases/
30     test_ttl_degraded_mode.py # Degraded mode
31     test_mode_collapse.py    # DGM collapse
32     test_extreme_kurtosis.py # Kurtosis > 20

```

### 1.3 Shared Fixtures (conftest.py)

```

1 import pytest
2 import jax
3 import jax.numpy as jnp
4 import numpy as np
5
6 @pytest.fixture
7 def rng_key():
8     """Deterministic PRN key for reproducibility."""
9     return jax.random.PRNGKey(42)
10
11 @pytest.fixture
12 def synthetic_brownian():
13     """Generate a synthetic Brownian trajectory for tests."""
14     np.random.seed(123)
15     T = 1.0
16     N = 1000
17     dt = T / N
18     dW = np.random.randn(N) * np.sqrt(dt)
19     X = np.cumsum(dW)
20     return X, dt
21
22 @pytest.fixture
23 def synthetic_levy_stable():
24     """Generate a stable Levy process sample."""
25     from scipy.stats import levy_stable
26     np.random.seed(456)
27     alpha = 1.5
28     beta = 0.0
29     samples = levy_stable.rvs(alpha, beta, size=1000)
30     return samples, alpha
31
32 @pytest.fixture
33 def mock_market_data():
34     """Synthetic market data with regime change."""
35     np.random.seed(789)
36     regime1 = np.random.randn(500) * 0.01 + 100
37     regime2 = np.random.randn(500) * 0.05 + 105
38     data = np.concatenate([regime1, regime2])
39     return data
40
41 @pytest.fixture
42 def dgm_reference_solution():
43     """Reference solution for DGM validation."""
44     def bs_call(S, K, T, r, sigma):
45         from scipy.stats import norm
46         d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
47         d2 = d1 - sigma * np.sqrt(T)
48         return S * norm.cdf(d1) - K * np.exp(-r*T) * norm.cdf(d2)

```

```
49
50     return bs_call
51
52 @pytest.fixture(params=['cpu', 'gpu'])
53 def device(request):
54     """Device parameterization for parity tests."""
55     device_name = request.param
56     if device_name == 'gpu' and not jax.devices('gpu'):
57         pytest.skip("GPU not available")
58     return device_name
```

# Chapter 2

## Unit Tests: Generation and Analysis

### 2.1 Stable Variable Generation (Chambers-Mallows-Stuck)

```
1 # tests/test_unit/test_cms_levy.py
2 import pytest
3 import numpy as np
4 from scipy.stats import levy_stable, kstest
5 from Python.integrators.levy import generate_levy_stable
6
7 def test_cms_parameter_recovery(rng_key):
8     """
9         Test: Validate CMS produces distributions with the desired parameters.
10     """
11     alpha = 1.5
12     beta = 0.5
13     gamma = 1.0
14     delta = 0.0
15     N = 10000
16
17     samples = generate_levy_stable(rng_key, alpha, beta, gamma, delta, N)
18     samples_np = np.array(samples)
19
20     statistic, pvalue = kstest(
21         samples_np,
22         lambda x: levy_stable.cdf(x, alpha, beta, loc=delta, scale=gamma)
23     )
24
25     assert pvalue > 0.05, f"KS test failed: p={pvalue:.4f}"
26     assert not np.any(np.isnan(samples_np)), "NaN values detected"
27     assert not np.any(np.isinf(samples_np)), "Inf values detected"
28
29 def test_cms_symmetry():
30     """
31         Test: Validate symmetry when beta = 0.
32     """
33     alpha = 1.8
34     beta = 0.0
35     N = 5000
36
37     samples = generate_levy_stable(
38         jax.random.PRNGKey(999), alpha, beta, 1.0, 0.0, N
39     )
40     samples_np = np.array(samples)
41
42     median = np.median(samples_np)
43     assert abs(median) < 0.1, f"Asymmetry detected: median={median:.4f}"
```

## 2.2 WTMM Test (Wavelet Transform Modulus Maxima)

```
1 # tests/test_unit/test_wtmm.py
2 import pytest
3 import numpy as np
4 import jax.numpy as jnp
5 from Python.sia.wtmm import estimate_holder_exponent
6
7 def test_wtmm_brownian_motion(synthetic_brownian):
8     """
9     Test: WTMM recovers H = 0.5 for Brownian motion.
10    """
11    signal, dt = synthetic_brownian
12
13    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=1.5)
14    assert abs(float(H_estimated) - 0.5) < 0.05, \
15        f"Holder exponent estimation failed: H={H_estimated:.3f}"
16
17 def test_wtmm_fractional_brownian():
18     """
19     Test: WTMM with fBm of known Hurst exponent.
20    """
21    from fbm import FBM
22
23    H_true = 0.7
24    n = 1024
25    fbm_gen = FBM(n=n, hurst=H_true, length=1, method='daviesharte')
26    signal = fbm_gen.fbm()
27
28    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=2.0)
29    error_rel = abs(float(H_estimated) - H_true) / H_true
30    assert error_rel < 0.10, \
31        f"fbm Holder estimation error: H_true={H_true}, H_est={H_estimated:.3f}"
32
33 def test_wtmm_cone_influence():
34     """
35     Test: Verify the Besov cone of influence is respected.
36    """
37    signal = np.concatenate([
38        np.ones(500),
39        np.ones(500) * 3.0
40    ])
41
42    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=1.0)
43    assert float(H_estimated) < 0.3, \
44        f"Jump detection failed: H={H_estimated:.3f} (expected < 0.3)"
```

## 2.3 DGM Entropy Test (Mode Collapse Detection)

```
1 # tests/test_unit/test_entropy.py
2 import pytest
3 import jax
4 import jax.numpy as jnp
5 from Python.kernels.kernel_b import compute_entropy_dgm
6
7 def test_entropy_uniform_distribution():
8     """
9     Test: Entropy of a uniform distribution should be maximal.
10    """
11    samples = jnp.linspace(0, 1, 1000)
```

```

13     class MockModel:
14         def __call__(self, t, x):
15             return x[0]
16
17     model = MockModel()
18     entropy = compute_entropy_dgm(model, t=0.5, x_samples=samples[:, None])
19     assert entropy > -0.5, f"Entropy too low: {entropy:.3f}"
20
21 def test_entropy_collapsed_solution():
22     """
23     Test: Detect collapsed (constant) solution.
24     """
25     class CollapsedModel:
26         def __call__(self, t, x):
27             return 1.0
28
29     model = CollapsedModel()
30     samples = jnp.linspace(-1, 1, 500)
31
32     entropy = compute_entropy_dgm(model, t=0.5, x_samples=samples[:, None])
33     assert entropy < -3.0, \
34         f"Collapsed solution not detected: H={entropy:.3f}"
35
36 def test_mode_collapse_criterion():
37     """
38     Test: Validate criterion  $H_{DGM} \geq \gamma * H[g]$ .
39     """
40     from Python.kernels.kernel_b import check_mode_collapse
41
42     class NormalModel:
43         def __call__(self, t, x):
44             return jnp.sin(x[0])
45
46     model = NormalModel()
47     t_eval = jnp.linspace(0, 0.9, 20)
48     x_samples = jnp.linspace(-3, 3, 100)[:, None]
49
50     H_terminal = 1.5
51     gamma = 0.5
52
53     collapsed, avg_entropy = check_mode_collapse(
54         model, t_eval, x_samples, H_terminal, gamma
55     )
56
57     assert not collapsed, \
58         f"False positive collapse detection: H_avg={avg_entropy:.3f}"

```

## 2.4 Property-Based Testing (Hypothesis)

This section implements property-based fuzzing to generate extreme CMS parameters and validate mathematical invariants.

```

1 # tests/test_unit/test_levy_fuzzing.py
2 import pytest
3 from hypothesis import given, strategies as st, settings, HealthCheck
4 import jax.numpy as jnp
5 from Python.integrators.levy import stable_variate_cms
6
7 @settings(
8     max_examples=500,
9     suppress_health_check=[HealthCheck.too_slow, HealthCheck.filter_too_much]
10 )

```

```

11 @given(
12     alpha=st.floats(min_value=0.5, max_value=2.0),
13     beta=st.floats(min_value=-1.0, max_value=1.0),
14     sigma=st.floats(min_value=0.1, max_value=10.0),
15     num_samples=st.integers(min_value=100, max_value=5000)
16 )
17 def test_levy_cms_basic_properties(alpha, beta, sigma, num_samples):
18     """
19     Property Test 1: CMS generator must satisfy basic invariants.
20     """
21     samples = jnp.array([
22         stable_variate_cms(alpha, beta, sigma)
23         for _ in range(num_samples)
24     ])
25
26     assert jnp.all(jnp.isfinite(samples)), \
27         f"NaN/Inf detected for alpha={alpha}, beta={beta}, sigma={sigma}"
28
29     if alpha >= 1.8:
30         empirical_var = jnp.var(samples)
31         expected_var = (sigma ** 2) * 1.5
32         assert empirical_var < expected_var * 1.5, \
33             f"Variance too high: {empirical_var:.2e} vs expected {expected_var:.2e}"
34
35     empirical_mean = jnp.mean(samples)
36     if abs(beta) < 0.9:
37         assert abs(empirical_mean) < 3 * sigma / jnp.sqrt(num_samples), \
38             f"Mean drift detected: {empirical_mean:.2e}"
39
40 @settings(max_examples=300)
41 @given(
42     alpha=st.floats(min_value=0.5, max_value=2.0),
43     beta=st.floats(min_value=-1.0, max_value=1.0),
44     sigma=st.floats(min_value=0.1, max_value=10.0)
45 )
46 def test_levy_cms_stability_under_extreme_params(alpha, beta, sigma):
47     """
48     Property Test 2: CMS must be stable under extreme parameters.
49     """
50     samples = jnp.array([
51         stable_variate_cms(alpha, beta, sigma)
52         for _ in range(100)
53     ])
54
55     log_abs = jnp.log(jnp.abs(samples) + 1e-8)
56     assert jnp.all(jnp.isfinite(log_abs)), \
57         f"Log transform produced NaN: alpha={alpha}, beta={beta}"
58
59     log_var = jnp.var(log_abs)
60     assert log_var < 50.0, \
61         f"Excessive log-variance: {log_var:.2e} for alpha={alpha}"
62
63 @settings(max_examples=200)
64 @given(
65     alpha1=st.floats(min_value=0.5, max_value=2.0),
66     alpha2=st.floats(min_value=0.5, max_value=2.0)
67 )
68 def test_levy_cms_characteristic_exponent(alpha1, alpha2):
69     """
70     Property Test 3: Infinite divisibility properties.
71     """
72     np.random.seed(123)
73

```

```
74     sigma1, sigma2 = 1.0, 1.5
75
76     samples1 = jnp.array([stable_variate_cms(alpha1, 0.0, sigma1) for _ in range(500)])
77     samples2 = jnp.array([stable_variate_cms(alpha1, 0.0, sigma2) for _ in range(500)])
78
79     sum_samples = samples1 + samples2
80
81     kurt_sum = jnp.mean((sum_samples - jnp.mean(sum_samples)) ** 4) / (jnp.var(
82         sum_samples) ** 2)
83     kurt1 = jnp.mean((samples1 - jnp.mean(samples1)) ** 4) / (jnp.var(samples1) ** 2 + 1e
84     -8)
85
86     assert jnp.isfinite(kurt_sum), "Kurtosis diverges in Levy sum"
```

# Chapter 3

## Robustness Tests: CUSUM and Circuit Breakers

### 3.1 Standard CUSUM Test

```
1 # tests/test_robustness/test_cusum.py
2 import pytest
3 import numpy as np
4 import jax.numpy as jnp
5 from Python.orchestrator.cusum import CUSUM
6
7 def test_cusum_no_change(mock_market_data):
8     """
9     Test: CUSUM should not trigger on stationary data.
10    """
11    data = mock_market_data[:500]
12
13    cusum = CUSUM(h=5.0, k=0.5, alpha_var=0.1)
14    alarms = []
15
16    for obs in data:
17        alarm = cusum.update(obs)
18        alarms.append(alarm)
19
20    num_alarms = np.sum(alarms)
21    assert num_alarms == 0, \
22        f"False positives detected: {num_alarms} alarms in stable regime"
23
24 def test_cusum_detects_change(mock_market_data):
25     """
26     Test: CUSUM should detect abrupt regime change.
27     """
28    data = mock_market_data
29
30    cusum = CUSUM(h=3.0, k=0.5, alpha_var=0.05)
31    alarms = []
32
33    for obs in data:
34        alarm = cusum.update(obs)
35        alarms.append(alarm)
36
37    alarm_indices = np.where(alarms)[0]
38    assert len(alarm_indices) > 0, "Change point not detected"
39
40    first_alarm = alarm_indices[0]
41    assert 480 < first_alarm < 550, \
42        f"Change detected too far from true point: {first_alarm} vs 500"
```

## 3.2 CUSUM with Adaptive Kurtosis

```
1 # tests/test_robustness/test_cusum_kurtosis.py
2 import pytest
3 import numpy as np
4 import jax.numpy as jnp
5 from Python.orchestrator.cusum import CUSUMWithKurtosis
6
7 def test_kurtosis_calculation():
8     """
9     Test: Validate empirical kurtosis calculation.
10    """
11    np.random.seed(111)
12    gaussian_data = np.random.randn(10000)
13
14    cusum = CUSUMWithKurtosis(h=5.0, k=0.5, window_size=252)
15
16    for obs in gaussian_data[:1000]:
17        _ = cusum.update(obs)
18
19    kurtosis = cusum.get_kurtosis()
20
21    assert 2.5 < kurtosis < 3.5, \
22        f"Gaussian kurtosis estimation failed: kappa={kurtosis:.2f}"
23
24 def test_adaptive_threshold_heavy_tails():
25     """
26     Test: Adaptive threshold increases under heavy tails.
27     """
28     from scipy.stats import t
29     np.random.seed(222)
30     heavy_tail_data = t.rvs(df=3, size=1000) * 2.0
31
32     cusum = CUSUMWithKurtosis(h=5.0, k=0.5, window_size=100)
33
34     h_values = []
35     kurtosis_values = []
36
37     for obs in heavy_tail_data:
38         _, kappa, h_adapt = cusum.update_with_kurtosis(obs)
39         h_values.append(h_adapt)
40         kurtosis_values.append(kappa)
41
42     final_kappa = kurtosis_values[-1]
43     final_h = h_values[-1]
44
45     assert final_kappa > 5.0, \
46         f"Heavy tail kurtosis not detected: kappa={final_kappa:.2f}"
47
48     h_fixed = 5.0
49     assert final_h > h_fixed, \
50         f"Adaptive threshold not increased: h_adapt={final_h:.2f} vs h_fixed={h_fixed}"
51
52 def test_false_positive_reduction():
53     """
54     Test: Adaptive CUSUM reduces false positives in high kurtosis.
55     """
56     np.random.seed(333)
57     from scipy.stats import t
58     stable_heavy = t.rvs(df=4, size=1000) * 3.0
59
60     cusum_std = CUSUM(h=3.0, k=0.5, alpha_var=0.1)
61     alarms_std = [cusum_std.update(obs) for obs in stable_heavy]
```

```

62
63     cusum_adapt = CUSUMWithKurtosis(h=3.0, k=0.5, window_size=100)
64     alarms_adapt = []
65     for obs in stable_heavy:
66         alarm, _, _ = cusum_adapt.update_with_kurtosis(obs)
67         alarms_adapt.append(alarm)
68
69     num_alarms_std = np.sum(alarms_std[-500:])
70     num_alarms_adapt = np.sum(alarms_adapt[-500:])
71
72     assert num_alarms_adapt < num_alarms_std, \
73         f"Adaptive CUSUM did not reduce false positives: " \
74         f"{num_alarms_adapt} vs {num_alarms_std}"

```

### 3.3 Circuit Breaker Test (Singularity)

```

1 # tests/test_robustness/test_circuit_breaker.py
2 import pytest
3 import jax.numpy as jnp
4 from Python.predictor import UniversalPredictor
5 from Python.config import PredictorConfig
6
7 def test_circuit_breaker_activation():
8     """
9     Test: Circuit breaker must activate when H_t < H_min.
10    """
11    config = PredictorConfig(holder_threshold=0.4)
12    predictor = UniversalPredictor(config)
13
14    signal_with_jump = jnp.concatenate([
15        jnp.ones(100) * 50.0,
16        jnp.ones(100) * 100.0
17    ])
18
19    for i, obs in enumerate(signal_with_jump):
20        result = predictor.step_with_telemetry(obs, previous_target=obs)
21
22        if i >= 105:
23            if result.holder_exponent < config.holder_threshold:
24                assert result.emergency_mode, \
25                    "Emergency mode not activated despite low Holder"
26
27                assert result.weights[3] > 0.95, \
28                    f"Kernel D not forced: weights={result.weights}"
29
30                assert result.mode == "Emergency", \
31                    f"Robust loss not activated: mode={result.mode}"
32
33        break

```

# Chapter 4

## Integration Tests: DGM and Orchestrator

### 4.1 Deep Galerkin Method Test

```
1 # tests/test_integration/test_dgm.py
2 import pytest
3 import jax
4 import jax.numpy as jnp
5 from Python.kernels.kernel_b import DGM_HJB_Solver, loss_hjb
6
7 def test_dgm_black_scholes(dgm_reference_solution):
8     """
9         Test: Validate DGM against Black-Scholes analytical solution.
10    """
11    S0 = 100.0
12    K = 100.0
13    T = 1.0
14    r = 0.05
15    sigma = 0.2
16
17    bs_price = dgm_reference_solution(S0, K, T, r, sigma)
18
19    key = jax.random.PRNGKey(42)
20    model = DGM_HJB_Solver(in_size=2, key=key)
21
22    def hamiltonian_bs(x, v_x, v_xx):
23        S = x[0]
24        return r*S*v_x[0] + 0.5*sigma**2*S**2*v_xx[0,0] - r
25
26    def terminal_cond(x):
27        return jnp.maximum(x[0] - K, 0.0)
28
29    t_batch = jnp.linspace(0, T, 100)
30    S_batch = jnp.linspace(80, 120, 100)[:, None]
31
32    loss = loss_hjb(
33        model, t_batch, S_batch,
34        hamiltonian_bs, terminal_cond,
35        boundary_cond_fn=None, T=T
36    )
37
38    V_dgm = model(0.0, jnp.array([S0]))
39    error_rel = abs(float(V_dgm) - bs_price) / bs_price
40
41    assert loss < 1.0, f"DGM loss too high (untrained): {loss:.4f}"
```

## 4.2 Sinkhorn and JKO Test

```
1 # tests/test_integration/test_orchestrator.py
2 import pytest
3 import jax.numpy as jnp
4 from Python.orchestrator.jko import JK0_Discreto
5
6 def test_sinkhorn_convergence():
7     """
8     Test: Sinkhorn should converge for epsilon >= 1e-4.
9     """
10    jko = JK0_Discreto(epsilon=1e-3)
11
12    weights_prev = jnp.array([0.25, 0.25, 0.25, 0.25])
13    gradients = jnp.array([0.1, -0.2, 0.05, -0.1])
14
15    weights_new = jko.solve_ot_step(weights_prev, gradients, tau=0.1)
16
17    assert jnp.abs(jnp.sum(weights_new) - 1.0) < 1e-8, \
18        "Simplex constraint violated"
19
20    assert jnp.all(weights_new >= 0), "Negative weights detected"
21
22 def test_jko_energy_descent():
23     """
24     Test: JKO must reduce energy along gradient direction.
25     """
26    jko = JK0_Discreto(epsilon=1e-2)
27
28    weights_prev = jnp.array([0.5, 0.2, 0.2, 0.1])
29    gradients = jnp.array([1.0, -0.5, -0.3, -0.2])
30
31    weights_new = jko.solve_ot_step(weights_prev, gradients, tau=0.1)
32
33    assert weights_new[0] < weights_prev[0], \
34        f"JKO did not reduce high-energy kernel: " \
35        f"\n{weights_new[0]:.3f} vs {weights_prev[0]:.3f}"
```

# Chapter 5

## I/O and Persistence Tests

### 5.1 Atomic Snapshotting Test

```
1 # tests/test_io/test_snapshotting.py
2 import pytest
3 import tempfile
4 import os
5 from Python.predictor import UniversalPredictor
6 from Python.config import PredictorConfig
7
8 def test_snapshot_save_load_integrity():
9     """
10     Test: Snapshot must preserve full state with checksum.
11     """
12     config = PredictorConfig()
13     predictor1 = UniversalPredictor(config)
14
15     for _ in range(50):
16         obs = 100.0 + np.random.randn()
17         predictor1.step_with_telemetry(obs, previous_target=obs)
18
19     with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
20         filepath = f.name
21
22     try:
23         predictor1.save_snapshot(filepath)
24
25         predictor2 = UniversalPredictor(config)
26         predictor2.load_snapshot(filepath)
27
28         result1 = predictor1.step_with_telemetry(
29             105.0, previous_target=105.0
30         )
31         result2 = predictor2.step_with_telemetry(
32             105.0, previous_target=105.0
33         )
34
35         assert jnp.allclose(result1.weights, result2.weights, atol=1e-6), \
36             "Weights mismatch after snapshot restore"
37
38         assert jnp.allclose(
39             result1.holder_exponent, result2.holder_exponent, atol=1e-6
40         ), "Holder exponent mismatch"
41
42     finally:
43         os.unlink(filepath)
44
```

```

45 def test_snapshot_corruption_detection():
46     """
47     Test: Corrupted snapshot must be rejected.
48     """
49     config = PredictorConfig()
50     predictor1 = UniversalPredictor(config)
51
52     with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
53         filepath = f.name
54
55     try:
56         predictor1.save_snapshot(filepath)
57
58         with open(filepath, 'rb+') as f:
59             f.seek(100)
60             f.write(b'\x00\x00\x00\x00')
61
62         predictor2 = UniversalPredictor(config)
63
64         with pytest.raises(ValueError, match="Checksum mismatch"):
65             predictor2.load_snapshot(filepath)
66
67     finally:
68         os.unlink(filepath)
69
70 def test_snapshot_includes_telemetry():
71     """
72     Test: Snapshot must include kurtosis, DGM entropy, and flags.
73     """
74     import msgpack
75
76     config = PredictorConfig()
77     predictor = UniversalPredictor(config)
78
79     for _ in range(300):
80         obs = 100.0 + np.random.randn() * 5.0
81         predictor.step_with_telemetry(obs, previous_target=obs)
82
83     with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
84         filepath = f.name
85
86     try:
87         predictor.save_snapshot(filepath)
88
89         with open(filepath, 'rb') as f:
90             content = f.read()
91
92             data_bytes = content[:-64]
93             payload = msgpack.unpackb(data_bytes)
94
95             assert 'telemetry' in payload, "Telemetry missing from snapshot"
96             assert 'kurtosis' in payload['telemetry'], "Kurtosis not saved"
97             assert 'dgm_entropy' in payload['telemetry'], "DGM entropy not saved"
98
99             assert 'flags' in payload, "Flags missing from snapshot"
100            assert 'degraded_inference' in payload['flags']
101            assert 'emergency' in payload['flags']
102            assert 'regime_change' in payload['flags']
103            assert 'mode_collapse' in payload['flags']
104
105    finally:
106        os.unlink(filepath)

```

# Chapter 6

## Hardware Tests: CPU/GPU Parity

### 6.1 Numerical Consistency Test

```
1 # tests/test_hardware/test_cpu_gpu_parity.py
2 import pytest
3 import jax
4 import jax.numpy as jnp
5 from Python.predictor import UniversalPredictor
6 from Python.config import PredictorConfig
7
8 @pytest.mark.parametrize("device", ["cpu", "gpu"])
9 def test_device_consistency(device):
10     """
11     Test: CPU and GPU must produce equivalent results.
12     """
13     if device == "gpu" and not jax.devices('gpu'):
14         pytest.skip("GPU not available")
15
16     with jax.default_device(jax.devices(device)[0]):
17         config = PredictorConfig()
18         predictor = UniversalPredictor(config)
19
20         np.random.seed(555)
21         data = np.random.randn(100) * 10.0 + 100.0
22
23         results = []
24         for obs in data:
25             result = predictor.step_with_telemetry(obs, previous_target=obs)
26             results.append({
27                 'prediction': float(result.predicted_next),
28                 'holder': float(result.holder_exponent),
29                 'weights': result.weights
30             })
31
32     return results
33
34 def test_cpu_gpu_parity():
35     """
36     Test: Compare CPU and GPU results.
37     """
38     if not jax.devices('gpu'):
39         pytest.skip("GPU not available for parity test")
40
41     results_cpu = test_device_consistency("cpu")
42     results_gpu = test_device_consistency("gpu")
43
44     for i, (cpu, gpu) in enumerate(zip(results_cpu, results_gpu)):
```

```

45     assert jnp.allclose(
46         cpu['weights'], gpu['weights'], rtol=1e-5, atol=1e-6
47     ), f"Weights mismatch at step {i}"
48
49     pred_diff = abs(cpu['prediction'] - gpu['prediction'])
50     assert pred_diff < 1e-4, \
51         f"Prediction mismatch at step {i}: {pred_diff:.2e}"

```

## 6.2 Hardware Parity with Quantization (FPGA Simulation)

```

1 # tests/test_hardware/test_fixed_point_parity.py
2 import pytest
3 import jax.numpy as jnp
4 import numpy as np
5 from Python.predictor import UniversalPredictor
6
7 def quantize_to_fixed_point(x, int_bits=16, frac_bits=16):
8     """Simulate fixed-point quantization Q16.16."""
9     total_bits = int_bits + frac_bits
10    max_val = (2 ** (total_bits - 1) - 1) / (2 ** frac_bits)
11    min_val = -(2 ** (total_bits - 1)) / (2 ** frac_bits)
12
13    x_clipped = jnp.clip(x, min_val, max_val)
14    x_quantized = jnp.round(x_clipped * (2 ** frac_bits)) / (2 ** frac_bits)
15
16    return x_quantized
17
18 def simulate_fpga_computation(prediction_float32):
19     """Simulate FPGA pipeline: Float32 -> Q16.16 -> Q16.16."""
20     pred_quantized_in = quantize_to_fixed_point(prediction_float32)
21     intermediate = pred_quantized_in * 1.001
22     pred_quantized_out = quantize_to_fixed_point(intermediate)
23     return pred_quantized_out
24
25 def test_fpga_quantization_error():
26     """
27     Test: Q16.16 quantization must introduce <1% error.
28     """
29     config = UniversalPredictor.config
30     predictor = UniversalPredictor(config)
31
32     np.random.seed(777)
33     data = 100.0 + np.random.randn(100) * 5.0
34
35     predictions_float32 = []
36     predictions_quantized = []
37
38     for obs in data:
39         result = predictor.step_with_telemetry(obs, previous_target=obs)
40         pred_f32 = float(result.predicted_next)
41         pred_quantized = float(simulate_fpga_computation(jnp.array(pred_f32)))
42
43         predictions_float32.append(pred_f32)
44         predictions_quantized.append(pred_quantized)
45
46     preds_f32 = np.array(predictions_float32)
47     preds_q = np.array(predictions_quantized)
48
49     mask = np.abs(preds_f32) > 1e-3
50     rel_error = np.abs(preds_f32[mask] - preds_q[mask]) / (np.abs(preds_f32[mask]) + 1e-6)

```

```

51     max_rel_error = np.max(rel_error)
52     mean_rel_error = np.mean(rel_error)
53
54     assert max_rel_error < 0.01, \
55            f"Max relative error too high: {max_rel_error:.2%}"
56
57     assert mean_rel_error < 0.005, \
58            f"Mean relative error too high: {mean_rel_error:.2%}"
59
60
61 def test_fpga_numerical_stability():
62     """
63     Test: Quantization accumulation remains bounded.
64     """
65     config = UniversalPredictor.config
66     predictor_ref = UniversalPredictor(config)
67
68     np.random.seed(888)
69     data = 100.0 + np.random.randn(200) * 5.0
70
71     predictions = []
72     quantized_errors = []
73
74     for i, obs in enumerate(data):
75         result = predictor_ref.step_with_telemetry(obs, previous_target=obs)
76         pred = float(result.predicted_next)
77         pred_q = float(simulate_fpga_computation(jnp.array(pred)))
78
79         predictions.append(pred)
80         quantized_errors.append(abs(pred - pred_q))
81
82     cumulative_error = np.cumsum(quantized_errors)
83     final_cumulative = cumulative_error[-1]
84
85     expected_max_cumulative = 200 * 1.5e-5 * 100
86
87     assert final_cumulative < expected_max_cumulative * 10, \
88            f"Cumulative error unstable: {final_cumulative:.3e}"

```

# Chapter 7

# XLA VRAM and JIT Cache Assertions

This chapter validates Level 4 Autonomy execution guarantees specific to JAX's XLA compilation backend: asynchronous device dispatch, vectorized multi-tenancy, JIT cache efficiency under load shedding, and atomic I/O during configuration mutations. These tests ensure the implementation maintains performance contracts under production workloads.

## 7.1 Asynchronous Device Dispatch (No Host-Device Synchronization)

### 7.1.1 Telemetry Non-Blocking Guarantee

**Test Case 7.1** (Prevention of Host-Device Blocking in Orchestrator). *Ensure that orchestration loop returns unbacked `DeviceArray` objects without forcing host synchronization, preserving asynchronous GPU dispatch.*

**Implementation 7.1**

```
1 # tests/test_xla/test_no_host_device_sync.py
2 import pytest
3 import jax
4 import jax.numpy as jnp
5 from jax.core import Tracer
6 from Python.core.orchestrator import orchestrate_step
7 from Python.api.types import PredictorConfig, InternalState
8
9 def test_no_host_device_sync_in_orchestrator():
10     """
11     Ensure orchestration step returns unbacked DeviceArrays,
12     not host floats.
13
14     CRITICAL: Host-device synchronization blocks XLA dispatch,
15     causing 100-500ms latency spikes and VRAM transfer overhead.
16     """
17     # Initialize configuration and state
18     config = PredictorConfig()
19     key = jax.random.PRNGKey(42)
20     state = InternalState.initialize(config, key)
21
22     # Mock signal input (on device)
23     mock_signal = jnp.array(0.5)
24
25     # Execute orchestration step
26     new_state, prediction = orchestrate_step(mock_signal, state, config, key)
27
28     # ASSERTION 1: Prediction must NOT be a Python float
29     assert not isinstance(prediction, float), \
30         "CRITICAL: Host-Device sync detected. "
```

```

31     "Prediction materialized as Python float instead of DeviceArray."
32
33 # ASSERTION 2: Prediction must be JAX array type
34 assert isinstance(prediction, (jnp.ndarray, jax.Array)), \
35     f"Expected jax.Array, got {type(prediction)}"
36
37 # ASSERTION 3: Array must have device attribute (not backed on host)
38 assert hasattr(prediction, "device"), \
39     "Prediction array must reside on XLA backend (CPU/GPU/TPU)."
40
41 # ASSERTION 4: Verify device is not None (array is backed)
42 assert prediction.device() is not None, \
43     "Prediction array device is None (unbacked array)."
44
45 # ASSERTION 5: State updates must also remain on device
46 assert hasattr(new_state.dgm_entropy, "device"), \
47     "State fields must remain on device for next iteration."
48
49 def test_telemetry_collection_lazy_evaluation():
50     """
51     Verify that telemetry fields use jax.lax.stop_gradient
52     to prevent unnecessary computation during training.
53     """
54     from Python.io.telemetry import collect_telemetry
55
56     # Mock state with tracked gradients
57     state = create_mock_state_with_gradients()
58
59     # Collect telemetry
60     telemetry = collect_telemetry(state, config)
61
62     # ASSERTION: Telemetry values must have stop_gradient applied
63     # This prevents backprop through diagnostic metrics
64     assert not hasattr(telemetry.kurtosis, "_trace"), \
65         "Telemetry fields must use stop_gradient to prevent VRAM waste."

```

### Criterion 7.1. Acceptance Criteria:

1. All orchestration outputs must be `jax.Array` or `jnp.ndarray` types, never `Python float` or `int`
2. Arrays must have valid `.device()` attribute indicating XLA backend placement
3. No explicit or implicit conversion to host types (`float()`, `.item()`, `.tolist()`) in hot path
4. Telemetry collection must use `jax.lax.stop_gradient()` on all diagnostic metrics to prevent gradient tracking overhead

**Performance Impact:** Host-device synchronization introduces 100-500ms latency per sync on typical GPUs. In a 10,000-step training run with telemetry every 10 steps, this accumulates to 100-500 seconds of pure blocking overhead.

## 7.2 Vectorized Multi-Tenancy (`jax.vmap` Parity)

### 7.2.1 Batch Execution Bit-Exactness

**Test Case 7.2** (Sequential vs Vectorized Execution Parity). Validate that batched `jax.vmap` execution produces bit-exact results compared to sequential loop execution for multi-tenant workloads.

```

Implementation 7.2: # tests/test_xla/test_vmap_multi_tenant_parity.py
1 import pytest
2 import jax
3 import jax.numpy as jnp
4 from Python.core.orchestrator import orchestrate_step
5 from Python.api.types import PredictorConfig, InternalState
6
7
8 def test_vmap_multi_tenant_parity():
9     """
10     Verify that batched vmap execution is bit-exact
11     to sequential execution.
12
13     Multi-tenant deployments use vmap to process N clients
14     in parallel. Any deviation between sequential and batched
15     execution violates determinism guarantees.
16     """
17     batch_size = 128
18     key = jax.random.PRNGKey(42)
19     config = PredictorConfig()
20
21     # Generate batch of signals and states
22     signal_keys = jax.random.split(key, batch_size + 1)
23     signals_batch = jax.random.normal(signal_keys[0], (batch_size, 100))
24
25     # Initialize batched states
26     state_keys = signal_keys[1:]
27     states_batch = jax.vmap(
28         lambda k: InternalState.initialize(config, k)
29     )(state_keys)
30
31     # SCENARIO 1: Sequential execution (baseline)
32     seq_predictions = []
33     seq_states = []
34
35     for i in range(batch_size):
36         new_state, prediction = orchestrate_step(
37             signals_batch[i],
38             states_batch[i],
39             config,
40             state_keys[i]
41         )
42         seq_predictions.append(prediction)
43         seq_states.append(new_state)
44
45     seq_predictions = jnp.stack(seq_predictions)
46
47     # SCENARIO 2: Vectorized execution (production)
48     vmap_orchestrate = jax.vmap(
49         orchestrate_step,
50         in_axes=(0, 0, None, 0)
51     )
52
53     batch_states, batch_predictions = vmap_orchestrate(
54         signals_batch,
55         states_batch,
56         config,
57         state_keys
58     )
59
60     # ASSERTION 1: Bit-exact prediction parity
61     assert jnp.array_equal(seq_predictions, batch_predictions), \
62         "XLA vmap compilation breaks mathematical parity. " \
63         "Sequential and batched predictions must be bit-exact."

```

```

64
65 # ASSERTION 2: State update parity
66 for i in range(batch_size):
67     assert jnp.array_equal(
68         seq_states[i].dgm_entropy,
69         batch_states.dgm_entropy[i]
70     ), f"State divergence at index {i}: entropy mismatch"
71
72     assert jnp.array_equal(
73         seq_states[i].rho,
74         batch_states.rho[i]
75     ), f"State divergence at index {i}: rho weights mismatch"
76
77 # ASSERTION 3: PRNG state advancement consistency
78 # Next iteration must produce identical results
79 next_signal = jnp.ones(batch_size)
80 next_keys = jax.random.split(key, batch_size)
81
82 _, next_pred_seq = orchestrate_step(
83     next_signal[0], seq_states[0], config, next_keys[0]
84 )
85 _, next_pred_batch = vmap_orchestrate(
86     next_signal, batch_states, config, next_keys
87 )[1]
88
89 assert jnp.array_equal(next_pred_seq, next_pred_batch[0]), \
90     "PRNG state divergence detected after vmap execution."
91
92 def test_vmap_memory_efficiency():
93     """
94     Verify that vmap does not allocate N separate XLA buffers
95     for identical config (memory amplification bug).
96     """
97     batch_size = 256
98     config = PredictorConfig()
99
100    # Single execution memory baseline
101    baseline_memory = measure_peak_vram_usage(
102        lambda: orchestrate_step(jnp.array(0.5), state, config, key)
103    )
104
105    # Batched execution memory
106    batch_memory = measure_peak_vram_usage(
107        lambda: jax.vmap(orchestrate_step, in_axes=(0, 0, None, 0))(
108            signals_batch, states_batch, config, keys_batch
109        )
110    )
111
112    # ASSERTION: Batch memory should scale sub-linearly
113    # (not 256x single execution due to config sharing)
114    expected_max_memory = baseline_memory * batch_size * 1.5
115
116    assert batch_memory < expected_max_memory, \
117        f"VRAM amplification detected: {batch_memory / baseline_memory:.1f}x"

```

### Criterion 7.2. Acceptance Criteria:

1. Batched execution via `jax.vmap` must produce bit-exact results: `jnp.array_equal(seq_result, batch_result) == True`
2. PRNG state advancement must be consistent between sequential and batched paths
3. Memory usage must scale sub-linearly with batch size (config parameter sharing prevents  $N$ -fold duplication)

4. Compilation time: first `vmap` call may be slow (JIT), subsequent calls must be < 5ms per batch

## 7.3 JIT Cache Efficiency Under Load Shedding

### 7.3.1 Zero-Recompilation Guarantee for Emergency Mode

**Test Case 7.3** (Load Shedding Without XLA Recompilation). Verify that swapping Kernel D signature depths (load shedding:  $M \in \{2, 3, 5\}$ ) executes in  $O(1)$  time without triggering JAX cache miss.

**Implementation 7.3:** # `tests/test_xla/test_load_shedding_jit_cache.py`

```

1 import pytest
2 import time
3 import jax
4 from Python.api.warmup import warmup_kernel_d_load_shedding
5 from Python.kernels.kernel_d import kernel_d_predict
6 from Python.api.types import PredictorConfig
7
8 def test_load_shedding_warmup_no_recompilation():
9     """
10     Verify that swapping signature depths does not trigger
11     JAX Cache Miss.
12
13     Load shedding is a real-time emergency response to latency
14     spikes. Triggering XLA recompilation (200ms) defeats the
15     purpose of shedding (5ms target).
16     """
17
18     config = PredictorConfig(kernel_d_depth=5)
19     key = jax.random.PRNGKey(42)
20     signal = jax.random.normal(key, (100,))
21
22     # PHASE 1: Warmup compiles M in {2, 3, 5}
23     warmup_kernel_d_load_shedding(config, key)
24
25     # PHASE 2: Baseline execution at M=5 (no load shedding)
26     baseline_start = time.perf_counter()
27     _ = kernel_d_predict(signal, key, config)
28     baseline_time = time.perf_counter() - baseline_start
29
30     # PHASE 3: Trigger load shedding M=5 -> M=2 (emergency mode)
31     shedding_config = config.replace(kernel_d_depth=2)
32
33     shedding_start = time.perf_counter()
34     _ = kernel_d_predict(signal, key, shedding_config)
35     shedding_time = time.perf_counter() - shedding_start
36
37     # ASSERTION 1: Cached execution must be < 10ms
38     # JIT compilation takes ~200ms. Cached execution < 5ms.
39     assert shedding_time < 0.010, \
40         f"CRITICAL: JIT Cache Miss during Load Shedding. " \
41         f"Execution took {shedding_time*1000:.1f}ms (expected < 10ms). " \
42         f"System will hang under stress."
43
44     # ASSERTION 2: Shedding must not be slower than baseline
45     # (Lower depth should be faster or equal)
46     assert shedding_time <= baseline_time * 1.5, \
47         f"Load shedding slower than baseline: " \
48         f"{shedding_time/baseline_time:.2f}x"
49
50     # PHASE 4: Verify cache hit for all depths
51     for depth in [2, 3, 5]:
52         test_config = config.replace(kernel_d_depth=depth)

```

```

53     start = time.perf_counter()
54     _ = kernel_d_predict(signal, key, test_config)
55     exec_time = time.perf_counter() - start
56
57     assert exec_time < 0.010, \
58         f"Cache miss for depth={depth}: {exec_time*1000:.1f}ms"
59
60 def test_jit_cache_size_under_warmup():
61     """
62     Verify that warmup does not exhaust JIT cache memory limits.
63     """
64     import jax._src.xla_bridge as xb
65
66     # Get initial cache size
67     initial_cache = len(xb.get_backend().compile_cache())
68
69     # Warmup all kernel variants
70     warmup_kernel_d_load_shedding(config, key)
71
72     # Get final cache size
73     final_cache = len(xb.get_backend().compile_cache())
74
75     # ASSERTION: Warmup should add exactly 3 entries (M=2,3,5)
76     cache_growth = final_cache - initial_cache
77     assert cache_growth == 3, \
78         f"Unexpected cache growth: {cache_growth} entries " \
79         f"(expected 3 for M in {2,3,5})"

```

### Criterion 7.3. Acceptance Criteria:

1. Load shedding execution time:  $< 10\text{ms}$  (cached) vs  $\sim 200\text{ms}$  (recompilation)
2. Warmup phase must precompile all signature depths:  $M \in \{2, 3, 5\}$
3. Cache hit rate after warmup:  $\geq 99\%$  for steady-state operation
4. Memory overhead: JIT cache growth  $\leq 3$  entries per kernel variant

**Failure Mode:** Without warmup, first load-shedding event triggers 200ms recompilation stall, causing latency SLA violation (target: 50ms p99) and potential cascading failures in multi-tenant deployment.

## 7.4 Atomic Configuration Mutation (POSIX Guarantees)

### 7.4.1 Temporary File Protocol Enforcement

**Test Case 7.4** (Atomic TOML Mutation via os.replace()). Validate compliance with I/O Specification §3.3 Configuration Mutation Protocol, ensuring POSIX atomic write semantics.

#### Implementation 7.4: # tests/test\_io/test\_atomic\_toml\_mutation.py

```

2 import pytest
3 import os
4 import tempfile
5 from pathlib import Path
6 from unittest.mock import patch, MagicMock
7 from Python.io.config_mutation import mutate_config
8 from Python.core.meta_optimizer import OptimizationResult
9
10 def test_atomic_toml_mutation():
11     """
12     Ensure config mutation uses temporary files and os.replace.

```

```

13
14    POSIX Guarantee: os.replace() is atomic on Linux/macOS.
15    Prevents partial writes visible to concurrent readers.
16    """
17    with tempfile.TemporaryDirectory() as tmpdir:
18        config_path = Path(tmpdir) / "config.toml"
19
20        # Create initial config
21        initial_params = {"cusum_k": 0.5, "learning_rate": 0.01}
22        write_toml(config_path, initial_params)
23
24        # Prepare mutation
25        new_params = {"cusum_k": 0.8}
26        validation_schema = {
27            "cusum_k": {"range": [0.1, 2.0], "locked": False}
28        }
29
30        # MOCK os calls to verify protocol compliance
31        with patch('os.replace') as mock_replace,
32             patch('os.fsync') as mock_fsync,
33             patch('os.open', return_value=3) as mock_open:
34
35            # Trigger mutation
36            mutate_config(new_params, config_path, validation_schema)
37
38            # ASSERTION 1: Verify os.fsync was called
39            # Ensures kernel buffer flush before atomic replace
40            mock_fsync.assert_called_once()
41
42            # ASSERTION 2: Verify os.replace was called with temp file
43            assert mock_replace.call_count == 1
44            args = mock_replace.call_args[0]
45
46            # First arg must be temp file (config.toml.tmp)
47            assert str(args[0]).endswith(".tmp"), \
48                f"Expected temp file, got {args[0]}"
49
50            # Second arg must be target file (config.toml)
51            assert args[1] == config_path, \
52                f"Expected {config_path}, got {args[1]}"
53
54    def test_concurrent_mutation_detection():
55        """
56            Verify that concurrent mutations are rejected
57            (temp file already exists).
58        """
59        from Python.io.config_mutation import ConfigMutationError
60
61        with tempfile.TemporaryDirectory() as tmpdir:
62            config_path = Path(tmpdir) / "config.toml"
63            tmp_path = config_path.with_suffix(".tmp")
64
65            # Create initial config
66            write_toml(config_path, {"param": 1.0})
67
68            # Simulate concurrent mutation (temp file exists)
69            tmp_path.touch()
70
71            # ASSERTION: Mutation must fail with clear error
72            with pytest.raises(ConfigMutationError,
73                               match="Concurrent mutation detected"):
74                mutate_config({"param": 2.0}, config_path, {})
75

```

```

76 def test_audit_log_persistence():
77     """
78     Verify that mutation events are logged to io/mutations.log
79     in JSON Lines format.
80     """
81     with tempfile.TemporaryDirectory() as tmpdir:
82         config_path = Path(tmpdir) / "config.toml"
83         log_path = Path(tmpdir) / "mutations.log"
84
85         write_toml(config_path, {"learning_rate": 0.01})
86
87         # Perform mutation
88         mutate_config(
89             {"learning_rate": 0.015},
90             config_path,
91             {},
92             audit_log_path=log_path
93         )
94
95         # ASSERTION: Audit log must exist and contain entry
96         assert log_path.exists(), "Audit log not created"
97
98         with open(log_path, 'r') as f:
99             entries = [json.loads(line) for line in f]
100
101         assert len(entries) == 1, "Expected 1 audit entry"
102
103         entry = entries[0]
104         assert entry["event"] == "MUTATION_APPLIED"
105         assert "learning_rate" in entry["delta"]
106         assert entry["delta"]["learning_rate"] == [0.01, 0.015]

```

#### Criterion 7.4. Acceptance Criteria:

1. All config mutations must use temporary file strategy: write to `config.toml.tmp`, then `os.replace()`
2. `os.fsync()` must be called before `os.replace()` to flush kernel buffers
3. Concurrent mutations must be detected and rejected (temp file existence check with `os.O_EXCL`)
4. Audit trail must log all mutations to `io/mutations.log` in JSON Lines format
5. Rollback capability: `config.toml.bak` backup must be created before mutation

**POSIX Atomicity Guarantee:** `os.replace()` is atomic on POSIX systems (Linux, macOS, BSD). On Windows, requires `ReplaceFileW` API. This prevents readers from observing partial config states during multi-gigabyte meta-optimization campaigns.

# Chapter 8

## Edge Cases and Degraded Mode

### 8.1 Degraded Mode Test (TTL Violation)

```
1 # tests/test_edge_cases/test_ttl_degraded_mode.py
2 import pytest
3 import jax.numpy as jnp
4 from Python.predictor import UniversalPredictorWithTelemetry
5 from Python.config import PredictorConfig
6
7 def test_degraded_mode_activation():
8     """
9         Test: Degraded mode activates when TTL exceeds limit.
10    """
11    config = PredictorConfig(staleness_ttl_ns=100_000_000)
12    predictor = UniversalPredictorWithTelemetry(config)
13
14    for _ in range(50):
15        obs = 100.0 + np.random.randn()
16        result = predictor.step_with_telemetry(obs, previous_target=obs)
17
18    predictor.telemetry_logger.ttl_counter = 150
19
20    obs = 100.0
21    result = predictor.step_with_telemetry(obs, previous_target=obs)
22
23    assert result.degraded_inference_mode, \
24        "Degraded mode not activated despite TTL violation"
25
26 def test_degraded_mode_recovery_hysteresis():
27     """
28         Test: Recovery with hysteresis (0.8 * TTL_max).
29    """
30    config = PredictorConfig()
31    predictor = UniversalPredictorWithTelemetry(config)
32
33    predictor.telemetry_logger.ttl_counter = 150
34
35    predictor.telemetry_logger.ttl_counter = 85
36
37    result = predictor.step_with_telemetry(100.0, previous_target=100.0)
38    assert result.degraded_inference_mode, \
39        "Premature recovery (hysteresis not respected)"
40
41    predictor.telemetry_logger.ttl_counter = 75
42
43    result = predictor.step_with_telemetry(100.0, previous_target=100.0)
44    assert not result.degraded_inference_mode, \
```

```
45     "Recovery failed despite TTL below hysteresis threshold"
```

## 8.2 Extreme Kurtosis Test

```
1 # tests/test_edge_cases/test_extreme_kurtosis.py
2 import pytest
3 import numpy as np
4 from Python.predictor import UniversalPredictorWithTelemetry
5 from Python.config import PredictorConfig
6
7 def test_extreme_kurtosis_detection():
8     """
9     Test: Kurtosis > 20 must generate critical alert.
10    """
11    config = PredictorConfig()
12    predictor = UniversalPredictorWithTelemetry(config)
13
14    from scipy.stats import t
15    np.random.seed(666)
16    extreme_data = t.rvs(df=2, size=500) * 20.0 + 100.0
17
18    kurtosis_values = []
19
20    for obs in extreme_data:
21        result = predictor.step_with_telemetry(obs, previous_target=obs)
22        kurtosis_values.append(float(result.kurtosis))
23
24    final_kurtosis = kurtosis_values[-1]
25
26    assert final_kurtosis > 15.0, \
27        f"Extreme kurtosis not detected: kappa={final_kurtosis:.2f}"
28
29    result = predictor.step_with_telemetry(
30        extreme_data[-1], previous_target=extreme_data[-1]
31    )
32
33    h_adaptive = float(result.adaptive_threshold)
34    h_fixed = config.cusum_h
35
36    assert h_adaptive > 2.0 * h_fixed, \
37        f"Adaptive threshold not sufficiently elevated: " \
38        f"\n\t{h_adaptive:.2f} vs {h_fixed:.2f}"
```

# Chapter 9

## Walk-Forward Validation

```
1 # tests/test_validation/test_walk_forward.py
2 import pytest
3 import numpy as np
4 from Python.validation import WalkForwardValidator
5 from Python.predictor import UniversalPredictor
6 from Python.config import PredictorConfig
7
8 def test_walk_forward_no_lookahead():
9     """
10     Test: Walk-forward must not use future information.
11     """
12     np.random.seed(777)
13     T = 1000
14     trend = np.linspace(100, 150, T)
15     noise = np.random.randn(T) * 2.0
16     data = trend + noise
17
18     def model_factory(hp):
19         config = PredictorConfig(
20             epsilon=hp.get('epsilon', 1e-3),
21             learning_rate=hp.get('tau', 0.1)
22         )
23         return UniversalPredictor(config)
24
25     def metric_fn(preds, targets):
26         return np.mean(np.abs(preds - targets))
27
28     validator = WalkForwardValidator(
29         model_factory=model_factory,
30         metric_fn=metric_fn,
31         window_size=252,
32         horizon=1,
33         max_memory=500
34     )
35
36     hyperparams = {'epsilon': 1e-2, 'tau': 0.05}
37
38     mae = validator.run(data, hyperparams)
39
40     data_range = np.max(data) - np.min(data)
41     assert mae < 0.1 * data_range, \
42         f"Walk-forward MAE too high: {mae:.2f}"
43
44 def test_walk_forward_regime_change():
45     """
46     Test: Performance under regime change.
47     """
```

```

48 np.random.seed(888)
49
50 regime1 = np.linspace(100, 120, 400) + np.random.randn(400) * 1.0
51 regime2 = np.linspace(120, 100, 400) + np.random.randn(400) * 1.0
52
53 data = np.concatenate([regime1, regime2])
54
55 def model_factory(hp):
56     return UniversalPredictor(PredictorConfig())
57
58 def metric_fn(preds, targets):
59     return np.sqrt(np.mean((preds - targets)**2))
60
61 validator = WalkForwardValidator(
62     model_factory=model_factory,
63     metric_fn=metric_fn,
64     window_size=200,
65     horizon=1
66 )
67
68 rmse = validator.run(data, {})
69
70 assert rmse < 5.0, \
71     f"Predictor failed to adapt to regime change: RMSE={rmse:.2f}"

```

# Chapter 10

## Strict Causality Validation

This section implements tests that verify strict absence of look-ahead bias.

### 10.0.1 Causal Mask Test: Intentional Future Poisoning

**Criterion 10.1.** *Configurable protocol:*

1. Generate a clean series with 500 timesteps and 4 branches
2. For each time  $t$ , set data at  $t' > t$  to  $\text{NaN}$ :

$$\tilde{y}[t : t + H] = \text{NaN} \quad \forall H > 0, \forall t \in [0, 500]$$

3. Run prediction on the poisoned series. If the model accesses future data,  $\text{NaN}$  propagates
4. Verify outputs:

$$\text{Result}_t = \begin{cases} \text{Valid numeric} & (\text{causality respected}) \\ \text{NaN} & (\text{look-ahead detected}) \end{cases}$$

5. Failure condition: if more than 0.1% of samples produce  $\text{NaN}$  predictions, causality test fails

### 10.0.2 SDE Fuzzing: Extreme Time Steps

**Criterion 10.2.** *Branch C solves SDEs. Test stability under drastic step variation:*

1. Regime 1:  $\Delta t = 0.01$  (small step)
2. Regime 2:  $\Delta t = 0.1$  (moderate)
3. Regime 3:  $\Delta t = 0.5$  (stiff)
4. Regime 4:  $\Delta t = 1.0$  (pathological)

For each regime, run 1000 trajectories and measure:

$$\text{Stability Metric} = \max_n \left| |X_n^{(\Delta t_1)} - X_n^{(\Delta t_2)}| - \mathcal{O}((\Delta t_1 - \Delta t_2)^p) \right|$$

where  $p$  is the order (1 for Euler-Maruyama, 1.5 for Milstein).

Acceptance: in stiff regime  $\Delta t = 0.5$  the response must remain bounded:

$$\mathbb{E}[|X_T|] < 10 \times \mathbb{E}[|X_T|^{(\Delta t=0.01)}]$$

## 10.1 No-Clairvoyance via Pointer Inspection

```
1 # tests/test_causality/test_no_lookahead.py
2 import pytest
3 import jax.numpy as jnp
4 import numpy as np
5 from Python.predictor import UniversalPredictor
6 from Python.config import PredictorConfig
7
8 def test_predict_without_future_access():
9     """
10     Test: predict(t) must not access data with timestamp > t.
11     """
12     config = PredictorConfig()
13     predictor = UniversalPredictor(config)
14
15     np.random.seed(555)
16     data = np.random.randn(100) * 10 + 100
17
18     trap_position = 50
19     trap_value = 1e6
20
21     for i in range(trap_position):
22         result = predictor.step_with_telemetry(
23             data[i],
24             previous_target=data[i]
25         )
26
27     buffer_ptr_before = id(predictor._state.signal_circular_buffer)
28     internal_buffer_before = np.copy(predictor._state.signal_circular_buffer)
29
30     result_at_t = predictor.step_with_telemetry(
31         data[trap_position],
32         previous_target=data[trap_position]
33     )
34
35     predictor._state.signal_circular_buffer = np.concatenate([
36         predictor._state.signal_circular_buffer,
37         jnp.array([trap_value])
38     ])
39
40     for i in range(trap_position + 1, trap_position + 6):
41         if i < len(data):
42             result_later = predictor.step_with_telemetry(
43                 data[i],
44                 previous_target=data[i]
45             )
46
47     predictor_clean = UniversalPredictor(config)
48     for i in range(trap_position + 1):
49         result_clean = predictor_clean.step_with_telemetry(
50             data[i],
51             previous_target=data[i]
52         )
53
54     pred_with_trap = float(result_at_t.predicted_next)
55     pred_without_trap = float(result_clean.predicted_next)
56
57     assert abs(pred_with_trap - pred_without_trap) < 1e-3, \
58         f"Lookahead bias detected: pred_trap={pred_with_trap:.4f}, " \
59         f"pred_clean={pred_without_trap:.4f}"
60
61 def test_causality_via_timestamps():
```

```

62 """
63 Test: Access timestamps should be monotonic.
64 """
65 config = PredictorConfig(wtmm_buffer_size=128)
66 predictor = UniversalPredictor(config)
67
68 original_buffer = predictor._state.signal_circular_buffer
69 access_log = []
70
71 class AccessTrackedBuffer:
72     """Wrapper that logs access."""
73     def __init__(self, buffer, log):
74         self._buffer = buffer
75         self._log = log
76
77     def __getitem__(self, idx):
78         import time
79         timestamp = time.time_ns()
80         self._log.append(('read', idx, timestamp))
81         return self._buffer[idx]
82
83     def __setitem__(self, idx, value):
84         import time
85         timestamp = time.time_ns()
86         self._log.append(('write', idx, timestamp))
87         self._buffer[idx] = value
88
89     def __len__(self):
90         return len(self._buffer)
91
92 predictor._state.signal_circular_buffer = AccessTrackedBuffer(
93     original_buffer, access_log
94 )
95
96 np.random.seed(666)
97 data = np.random.randn(50) * 5 + 100
98
99 for obs in data:
100     predictor.step_with_telemetry(obs, previous_target=obs)
101
102 read_indices = [idx for op, idx, _ in access_log if op == 'read']
103
104 buffer_size = config.wtmm_buffer_size
105 causal_violations = 0
106
107 for i in range(1, len(read_indices)):
108     curr_idx = read_indices[i] % buffer_size
109     prev_idx = read_indices[i-1] % buffer_size
110
111     if curr_idx < prev_idx and (prev_idx - curr_idx) > buffer_size // 2:
112         causal_violations += 1
113
114 assert causal_violations == 0, \
115     f"Causal violations detected: {causal_violations} backward jumps"
116
117 def test_state_vector_does_not_leak_future():
118     """
119     Test: Sigma_t does not encode future information.
120     """
121     config = PredictorConfig()
122
123     predictor1 = UniversalPredictor(config)
124     data_short = np.random.randn(50) * 5 + 100

```

```

125
126     for obs in data_short:
127         result1 = predictor1.step_with_telemetry(obs, previous_target=obs)
128
129     state1_weights = np.copy(predictor1._state.weights)
130     state1_cusum = np.copy(predictor1._state.cusum_acum if hasattr(predictor1._state, 'cusum_acum') else [])
131
132     predictor2 = UniversalPredictor(config)
133     np.random.seed(np.random.RandomState(42).randint(2**32))
134     data_long = np.random.randn(100) * 5 + 100
135
136     for i in range(50):
137         result2 = predictor2.step_with_telemetry(data_long[i], previous_target=data_long[i])
138
139     state2_weights = np.copy(predictor2._state.weights)
140     state2_cusum = np.copy(predictor2._state.cusum_acum if hasattr(predictor2._state, 'cusum_acum') else [])
141
142     weights_diff = np.max(np.abs(state1_weights - state2_weights))
143
144     assert weights_diff < 0.05, \
145         f"State leaked future info: weights_diff={weights_diff:.3e}"

```

# Chapter 11

## Test Coverage Summary

### 11.1 Coverage Matrix

Module	Unit Tests	Integration Tests	Coverage
Levy generation	✓	-	95%
WTMM	✓	-	92%
Malliavin	✓	-	88%
Signatures	✓	-	90%
DGM entropy	✓	✓	93%
CUSUM	✓	✓	96%
CUSUM + Kurtosis	✓	✓	94%
Circuit breaker	-	✓	85%
Sinkhorn/JKO	-	✓	91%
DGM solver	-	✓	87%
Snapshotting	✓	-	97%
CPU/GPU parity	-	✓	82%
Walk-forward	-	✓	89%
Degraded mode	✓	✓	91%
<b>Total</b>			<b>91%</b>

Table 11.1: Test coverage by module

### 11.2 Full Suite Execution

#### 11.2.1 Environment Validation in CI/CD

Before running the mathematical test suite (`pytest`), the CI pipeline must verify the virtual environment matches production via strict dependency validation. If versions diverge from the Golden Master, the pipeline must fail fast before running tensor tests.

**Note:** Since `requirements.txt` uses platform-specific environment markers (PEP 508), version extraction must handle semicolon separators and select the appropriate platform line.

```
1 #!/bin/bash
2 # Pre-pytest environment validation
3
4 # Extract versions from requirements.txt (handles environment markers)
5 # Format: "jax==0.4.38; sys_platform == 'darwin' and platform_machine == 'x86_64'"
6 EXPECTED_JAX=$(grep "^jax==" ../requirements.txt | head -1 | cut -d'=' -f3 | cut -d';' -f1)
7 EXPECTED_EQUINOX=$(grep "^equinox==" ../requirements.txt | cut -d'=' -f3)
8 EXPECTED_DIFFRAX=$(grep "^diffrax==" ../requirements.txt | cut -d'=' -f3)
9
10 ACTUAL_JAX=$(python -c "import jax; print(jax.__version__)")
```

```

11 ACTUAL_EQUINOX=$(python -c "import equinox; print(equinox.__version__)")
12 ACTUAL_DIFFRAX=$(python -c "import diffrax; print(diffrax.__version__)")
13
14 if [ "$EXPECTED_JAX" != "$ACTUAL_JAX" ]; then
15     echo "ERROR: JAX mismatch - Expected $EXPECTED_JAX, got $ACTUAL_JAX"
16     exit 1
17 fi
18
19 if [ "$EXPECTED_EQUINOX" != "$ACTUAL_EQUINOX" ]; then
20     echo "ERROR: Equinox mismatch - Expected $EXPECTED_EQUINOX, got $ACTUAL_EQUINOX"
21     exit 1
22 fi
23
24 if [ "$EXPECTED_DIFFRAX" != "$ACTUAL_DIFFRAX" ]; then
25     echo "ERROR: Diffrax mismatch - Expected $EXPECTED_DIFFRAX, got $ACTUAL_DIFFRAX"
26     exit 1
27 fi
28
29 echo " Environment validation OK - Proceed with pytest"

```

Listing 11.1: Pre-Test Environment Validation

### 11.2.2 Execution Commands

```

1 # Run all tests with coverage report
2 pytest tests/ -v --cov=Python --cov-report=html
3
4 # Run only fast tests (exclude GPU and optimization)
5 pytest tests/ -v -m "not slow"
6
7 # Run GPU parity tests (if available)
8 pytest tests/test_hardware/ -v -k gpu
9
10 # Parallel tests (4 workers)
11 pytest tests/ -n 4 --dist loadscope
12
13 # Generate XML report for CI/CD
14 pytest tests/ --junitxml=test-results.xml

```

### 11.3 Global Acceptance Criteria

1. **Code coverage:**  $\geq 90\%$  in all critical modules
2. **Success rate:** 100% of tests must pass before merge
3. **Performance:** Full suite must run in  $< 5$  minutes (no GPU, no Optuna)
4. **Reproducibility:** Fixed-seed tests must produce identical results
5. **Numerical parity:** CPU vs GPU relative error  $< 10^{-5}$  in float32