Python Test Suite for the Universal Stochastic Predictor

Adaptive Meta-Prediction Development Consortium

February 21, 2026

Contents

Chapter 1

Testing Environment Setup

1.1 Dependencies and Tools

```
# requirements-test.txt
pytest>=7.4.0
pytest-cov>=4.1.0
pytest-xdist>=3.3.0  # Test parallelization
hypothesis>=6.82.0   # Property-based testing
jax[cpu]>=0.4.13     # CPU tests
jax[cuda12]>=0.4.13  # GPU tests (optional)
numpy>=1.24.0
scipy>=1.11.0
PyWavelets>=1.4.1
msgpack>=1.0.5
optuna>=3.2.0

# Validation tools
flake8>=6.0.0
mypy>=1.4.0
black>=23.7.0
```

1.2 Directory Structure

```
tests/
 __init__.py
 conftest.py                    # Shared fixtures
 test_unit/
     test_cms_levy.py           # Stable variable generation
     test_wtmm.py               # Multifractal analysis
     test_malliavin.py          # Sensitivity computations
     test_signatures.py         # Branch D
     test_entropy.py            # DGM entropy
 test_integration/
     test_sde_solvers.py        # Euler-Maruyama, Milstein
     test_sinkhorn.py           # Optimal transport
     test_dgm.py                # Deep Galerkin Method
     test_orchestrator.py     # Full JKO
 test_robustness/
     test_cusum.py              # Change detection
     test_cusum_kurtosis.py     # CUSUM with kurtosis
     test_circuit_breaker.py    # Singularities
     test_outliers.py           # Extreme values
 test_io/
     test_snapshotting.py       # Persistence
     test_recovery.py           # Atomic recovery
```

```
test_hardware/
    test_cpu_gpu_parity.py    # Numerical consistency
    test_numerical_drift.py   # Fixed-point drift
test_validation/
    test_walk_forward.py      # Causal validation
    test_optuna_tuning.py     # Meta-optimization
test_edge_cases/
     test_ttl_degraded_mode.py # Degraded mode
     test_mode_collapse.py     # DGM collapse
     test_extreme_kurtosis.py  # Kurtosis > 20
```

1.3 Shared Fixtures (conftest.py)

```python
import pytest
import jax
import jax.numpy as jnp
import numpy as np

@pytest.fixture
def rng_key():
    """Deterministic PRN key for reproducibility."""
    return jax.random.PRNGKey(42)

@pytest.fixture
def synthetic_brownian():
    """Generate a synthetic Brownian trajectory for tests."""
    np.random.seed(123)
    T = 1.0
    N = 1000
    dt = T / N
    dW = np.random.randn(N) * np.sqrt(dt)
    X = np.cumsum(dW)
    return X, dt

@pytest.fixture
def synthetic_levy_stable():
    """Generate a stable Levy process sample."""
    from scipy.stats import levy_stable
    np.random.seed(456)
    alpha = 1.5
    beta = 0.0
    samples = levy_stable.rvs(alpha, beta, size=1000)
    return samples, alpha

@pytest.fixture
def mock_market_data():
    """Synthetic market data with regime change."""
    np.random.seed(789)
    regime1 = np.random.randn(500) * 0.01 + 100
    regime2 = np.random.randn(500) * 0.05 + 105
    data = np.concatenate([regime1, regime2])
    return data

@pytest.fixture
def dgm_reference_solution():
    """Reference solution for DGM validation."""
    def bs_call(S, K, T, r, sigma):
        from scipy.stats import norm
        d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
        d2 = d1 - sigma * np.sqrt(T)
        return S * norm.cdf(d1) - K * np.exp(-r*T) * norm.cdf(d2)

    return bs_call

@pytest.fixture(params=['cpu', 'gpu'])
def device(request):
    """Device parameterization for parity tests."""
    device_name = request.param
    if device_name == 'gpu' and not jax.devices('gpu'):
        pytest.skip("GPU not available")
    return device_name
```

Chapter 2

Unit Tests: Generation and Analysis

2.1 Stable Variable Generation (Chambers-Mallows-Stuck)

```python
# tests/test_unit/test_cms_levy.py
import pytest
import numpy as np
from scipy.stats import levy_stable, kstest
from Python.integrators.levy import generate_levy_stable

def test_cms_parameter_recovery(rng_key):
    """
    Test: Validate CMS produces distributions with the desired parameters.
    """
    alpha = 1.5
    beta = 0.5
    gamma = 1.0
    delta = 0.0
    N = 10000

    samples = generate_levy_stable(rng_key, alpha, beta, gamma, delta, N)
    samples_np = np.array(samples)

    statistic, pvalue = kstest(
        samples_np,
        lambda x: levy_stable.cdf(x, alpha, beta, loc=delta, scale=gamma)
    )

    assert pvalue > 0.05, f"KS test failed: p={pvalue:.4f}"
    assert not np.any(np.isnan(samples_np)), "NaN values detected"
    assert not np.any(np.isinf(samples_np)), "Inf values detected"

def test_cms_symmetry():
    """
    Test: Validate symmetry when beta = 0.
    """
    alpha = 1.8
    beta = 0.0
    N = 5000

    samples = generate_levy_stable(
        jax.random.PRNGKey(999), alpha, beta, 1.0, 0.0, N
    )
    samples_np = np.array(samples)

    median = np.median(samples_np)
    assert abs(median) < 0.1, f"Asymmetry detected: median={median:.4f}"
```

2.2 WTMM Test (Wavelet Transform Modulus Maxima)

```python
# tests/test_unit/test_wtmm.py
import pytest
import numpy as np
import jax.numpy as jnp
from Python.sia.wtmm import estimate_holder_exponent

def test_wtmm_brownian_motion(synthetic_brownian):
    """
    Test: WTMM recovers H   0.5 for Brownian motion.
    """
    signal, dt = synthetic_brownian

    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=1.5)
    assert abs(float(H_estimated) - 0.5) < 0.05, \
        f"Holder exponent estimation failed: H={H_estimated:.3f}"

def test_wtmm_fractional_brownian():
    """
    Test: WTMM with fBm of known Hurst exponent.
    """
    from fbm import FBM

    H_true = 0.7
    n = 1024
    fbm_gen = FBM(n=n, hurst=H_true, length=1, method='daviesharte')
    signal = fbm_gen.fbm()

    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=2.0)
    error_rel = abs(float(H_estimated) - H_true) / H_true
    assert error_rel < 0.10, \
        f"fBm Holder estimation error: H_true={H_true}, H_est={H_estimated:.3f}"

def test_wtmm_cone_influence():
    """
    Test: Verify the Besov cone of influence is respected.
    """
    signal = np.concatenate([
        np.ones(500),
        np.ones(500) * 3.0
    ])

    H_estimated = estimate_holder_exponent(jnp.array(signal), besov_c=1.0)
    assert float(H_estimated) < 0.3, \
        f"Jump detection failed: H={H_estimated:.3f} (expected < 0.3)"
```

2.3 DGM Entropy Test (Mode Collapse Detection)

```python
# tests/test_unit/test_entropy.py
import pytest
import jax
import jax.numpy as jnp
from Python.kernels.kernel_b import compute_entropy_dgm

def test_entropy_uniform_distribution():
    """
    Test: Entropy of a uniform distribution should be maximal.
    """
    samples = jnp.linspace(0, 1, 1000)

    class MockModel:
        def __call__(self, t, x):
            return x[0]

    model = MockModel()
    entropy = compute_entropy_dgm(model, t=0.5, x_samples=samples[:, None])
    assert entropy > -0.5, f"Entropy too low: {entropy:.3f}"

def test_entropy_collapsed_solution():
    """
    Test: Detect collapsed (constant) solution.
    """
    class CollapsedModel:
        def __call__(self, t, x):
            return 1.0

    model = CollapsedModel()
    samples = jnp.linspace(-1, 1, 500)

    entropy = compute_entropy_dgm(model, t=0.5, x_samples=samples[:, None])
    assert entropy < -3.0, \
        f"Collapsed solution not detected: H={entropy:.3f}"

def test_mode_collapse_criterion():
    """
    Test: Validate criterion H_DGM >= gamma * H[g].
    """
    from Python.kernels.kernel_b import check_mode_collapse

    class NormalModel:
        def __call__(self, t, x):
            return jnp.sin(x[0])

    model = NormalModel()
    t_eval = jnp.linspace(0, 0.9, 20)
    x_samples = jnp.linspace(-3, 3, 100)[:, None]

    H_terminal = 1.5
    gamma = 0.5

    collapsed, avg_entropy = check_mode_collapse(
        model, t_eval, x_samples, H_terminal, gamma
    )

    assert not collapsed, \
        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.

```python
# tests/test_unit/test_levy_fuzzing.py
import pytest
from hypothesis import given, strategies as st, settings, HealthCheck
import jax.numpy as jnp
from Python.integrators.levy import stable_variate_cms

@settings(
    max_examples=500,
    suppress_health_check=[HealthCheck.too_slow, HealthCheck.filter_too_much]
)
```

```
@given(
    alpha=st.floats(min_value=0.5, max_value=2.0),
    beta=st.floats(min_value=-1.0, max_value=1.0),
    sigma=st.floats(min_value=0.1, max_value=10.0),
    num_samples=st.integers(min_value=100, max_value=5000)
)
def test_levy_cms_basic_properties(alpha, beta, sigma, num_samples):
    """
    Property Test 1: CMS generator must satisfy basic invariants.
    """
    samples = jnp.array([
        stable_variate_cms(alpha, beta, sigma)
        for _ in range(num_samples)
    ])

    assert jnp.all(jnp.isfinite(samples)), \
        f"NaN/Inf detected for alpha={alpha}, beta={beta}, sigma={sigma}"

    if alpha >= 1.8:
        empirical_var = jnp.var(samples)
        expected_var = (sigma ** 2) * 1.5
        assert empirical_var < expected_var * 1.5, \
            f"Variance too high: {empirical_var:.2e} vs expected {expected_var:.2e}"

    empirical_mean = jnp.mean(samples)
    if abs(beta) < 0.9:
        assert abs(empirical_mean) < 3 * sigma / jnp.sqrt(num_samples), \
            f"Mean drift detected: {empirical_mean:.2e}"

@settings(max_examples=300)
@given(
    alpha=st.floats(min_value=0.5, max_value=2.0),
    beta=st.floats(min_value=-1.0, max_value=1.0),
    sigma=st.floats(min_value=0.1, max_value=10.0)
)
def test_levy_cms_stability_under_extreme_params(alpha, beta, sigma):
    """
    Property Test 2: CMS must be stable under extreme parameters.
    """
    samples = jnp.array([
        stable_variate_cms(alpha, beta, sigma)
        for _ in range(100)
    ])

    log_abs = jnp.log(jnp.abs(samples) + 1e-8)
    assert jnp.all(jnp.isfinite(log_abs)), \
        f"Log transform produced NaN: alpha={alpha}, beta={beta}"

    log_var = jnp.var(log_abs)
    assert log_var < 50.0, \
        f"Excessive log-variance: {log_var:.2e} for alpha={alpha}"

@settings(max_examples=200)
@given(
    alpha1=st.floats(min_value=0.5, max_value=2.0),
    alpha2=st.floats(min_value=0.5, max_value=2.0)
)
def test_levy_cms_characteristic_exponent(alpha1, alpha2):
    """
    Property Test 3: Infinite divisibility properties.
    """
    np.random.seed(123)

    sigma1, sigma2 = 1.0, 1.5

    samples1 = jnp.array([stable_variate_cms(alpha1, 0.0, sigma1) for _ in range(500)])
    samples2 = jnp.array([stable_variate_cms(alpha1, 0.0, sigma2) for _ in range(500)])

    sum_samples = samples1 + samples2

    kurt_sum = jnp.mean((sum_samples - jnp.mean(sum_samples)) ** 4) / (jnp.var(
    sum_samples) ** 2)
    kurt1 = jnp.mean((samples1 - jnp.mean(samples1)) ** 4) / (jnp.var(samples1) ** 2 + 1e
    -8)

    assert jnp.isfinite(kurt_sum), "Kurtosis diverges in Levy sum"
```

Chapter 3

Robustness Tests: CUSUM and Circuit Breakers

3.1 Standard CUSUM Test

```python
# tests/test_robustness/test_cusum.py
import pytest
import numpy as np
import jax.numpy as jnp
from Python.orchestrator.cusum import CUSUM

def test_cusum_no_change(mock_market_data):
    """
    Test: CUSUM should not trigger on stationary data.
    """
    data = mock_market_data[:500]

    cusum = CUSUM(h=5.0, k=0.5, alpha_var=0.1)
    alarms = []

    for obs in data:
        alarm = cusum.update(obs)
        alarms.append(alarm)

    num_alarms = np.sum(alarms)
    assert num_alarms == 0, \
        f"False positives detected: {num_alarms} alarms in stable regime"

def test_cusum_detects_change(mock_market_data):
    """
    Test: CUSUM should detect abrupt regime change.
    """
    data = mock_market_data

    cusum = CUSUM(h=3.0, k=0.5, alpha_var=0.05)
    alarms = []

    for obs in data:
        alarm = cusum.update(obs)
        alarms.append(alarm)

    alarm_indices = np.where(alarms)[0]
    assert len(alarm_indices) > 0, "Change point not detected"

    first_alarm = alarm_indices[0]
    assert 480 < first_alarm < 550, \
        f"Change detected too far from true point: {first_alarm} vs 500"
```

3.2 CUSUM with Adaptive Kurtosis

```python
# tests/test_robustness/test_cusum_kurtosis.py
import pytest
import numpy as np
import jax.numpy as jnp
from Python.orchestrator.cusum import CUSUMWithKurtosis

def test_kurtosis_calculation():
    """
    Test: Validate empirical kurtosis calculation.
    """
    np.random.seed(111)
    gaussian_data = np.random.randn(10000)

    cusum = CUSUMWithKurtosis(h=5.0, k=0.5, window_size=252)

    for obs in gaussian_data[:1000]:
        _ = cusum.update(obs)

    kurtosis = cusum.get_kurtosis()

    assert 2.5 < kurtosis < 3.5, \
        f"Gaussian kurtosis estimation failed: kappa={kurtosis:.2f}"

def test_adaptive_threshold_heavy_tails():
    """
    Test: Adaptive threshold increases under heavy tails.
    """
    from scipy.stats import t
    np.random.seed(222)
    heavy_tail_data = t.rvs(df=3, size=1000) * 2.0

    cusum = CUSUMWithKurtosis(h=5.0, k=0.5, window_size=100)

    h_values = []
    kurtosis_values = []

    for obs in heavy_tail_data:
        _, kappa, h_adapt = cusum.update_with_kurtosis(obs)
        h_values.append(h_adapt)
        kurtosis_values.append(kappa)

    final_kappa = kurtosis_values[-1]
    final_h = h_values[-1]

    assert final_kappa > 5.0, \
        f"Heavy tail kurtosis not detected: kappa={final_kappa:.2f}"

    h_fixed = 5.0
    assert final_h > h_fixed, \
        f"Adaptive threshold not increased: h_adapt={final_h:.2f} vs h_fixed={h_fixed}"

def test_false_positive_reduction():
    """
    Test: Adaptive CUSUM reduces false positives in high kurtosis.
    """
    np.random.seed(333)
    from scipy.stats import t
    stable_heavy = t.rvs(df=4, size=1000) * 3.0

    cusum_std = CUSUM(h=3.0, k=0.5, alpha_var=0.1)
    alarms_std = [cusum_std.update(obs) for obs in stable_heavy]

    cusum_adapt = CUSUMWithKurtosis(h=3.0, k=0.5, window_size=100)
    alarms_adapt = []
    for obs in stable_heavy:
        alarm, _, _ = cusum_adapt.update_with_kurtosis(obs)
        alarms_adapt.append(alarm)

    num_alarms_std = np.sum(alarms_std[-500:])
    num_alarms_adapt = np.sum(alarms_adapt[-500:])

    assert num_alarms_adapt < num_alarms_std, \
        f"Adaptive CUSUM did not reduce false positives: " \
        f"{num_alarms_adapt} vs {num_alarms_std}"
```

3.3 Circuit Breaker Test (Singularity)

```python
# tests/test_robustness/test_circuit_breaker.py
import pytest
import jax.numpy as jnp
from Python.predictor import UniversalPredictor
from Python.config import PredictorConfig

def test_circuit_breaker_activation():
    """
    Test: Circuit breaker must activate when H_t < H_min.
    """
    config = PredictorConfig(holder_threshold=0.4)
    predictor = UniversalPredictor(config)

    signal_with_jump = jnp.concatenate([
        jnp.ones(100) * 50.0,
        jnp.ones(100) * 100.0
    ])

    for i, obs in enumerate(signal_with_jump):
        result = predictor.step_with_telemetry(obs, previous_target=obs)

        if i >= 105:
            if result.holder_exponent < config.holder_threshold:
                assert result.emergency_mode, \
                    "Emergency mode not activated despite low Holder"

                assert result.weights[3] > 0.95, \
                    f"Kernel D not forced: weights={result.weights}"

                assert result.mode == "Emergency", \
                    f"Robust loss not activated: mode={result.mode}"

                break
```

Chapter 4

Integration Tests: DGM and Orchestrator

4.1 Deep Galerkin Method Test

```python
# tests/test_integration/test_dgm.py
import pytest
import jax
import jax.numpy as jnp
from Python.kernels.kernel_b import DGM_HJB_Solver, loss_hjb

def test_dgm_black_scholes(dgm_reference_solution):
    """
    Test: Validate DGM against Black-Scholes analytical solution.
    """
    S0 = 100.0
    K = 100.0
    T = 1.0
    r = 0.05
    sigma = 0.2

    bs_price = dgm_reference_solution(S0, K, T, r, sigma)

    key = jax.random.PRNGKey(42)
    model = DGM_HJB_Solver(in_size=2, key=key)

    def hamiltonian_bs(x, v_x, v_xx):
        S = x[0]
        return r*S*v_x[0] + 0.5*sigma**2*S**2*v_xx[0,0] - r

    def terminal_cond(x):
        return jnp.maximum(x[0] - K, 0.0)

    t_batch = jnp.linspace(0, T, 100)
    S_batch = jnp.linspace(80, 120, 100)[:, None]

    loss = loss_hjb(
        model, t_batch, S_batch,
        hamiltonian_bs, terminal_cond,
        boundary_cond_fn=None, T=T
    )

    V_dgm = model(0.0, jnp.array([S0]))
    error_rel = abs(float(V_dgm) - bs_price) / bs_price

    assert loss < 1.0, f"DGM loss too high (untrained): {loss:.4f}"
```

4.2 Sinkhorn and JKO Test

```python
# tests/test_integration/test_orchestrator.py
import pytest
import jax.numpy as jnp
from Python.orchestrator.jko import JKO_Discreto

def test_sinkhorn_convergence():
    """
    Test: Sinkhorn should converge for epsilon >= 1e-4.
    """
    jko = JKO_Discreto(epsilon=1e-3)

    weights_prev = jnp.array([0.25, 0.25, 0.25, 0.25])
    gradients = jnp.array([0.1, -0.2, 0.05, -0.1])

    weights_new = jko.solve_ot_step(weights_prev, gradients, tau=0.1)

    assert jnp.abs(jnp.sum(weights_new) - 1.0) < 1e-8, \
        "Simplex constraint violated"

    assert jnp.all(weights_new >= 0), "Negative weights detected"

def test_jko_energy_descent():
    """
    Test: JKO must reduce energy along gradient direction.
    """
    jko = JKO_Discreto(epsilon=1e-2)

    weights_prev = jnp.array([0.5, 0.2, 0.2, 0.1])
    gradients = jnp.array([1.0, -0.5, -0.3, -0.2])

    weights_new = jko.solve_ot_step(weights_prev, gradients, tau=0.1)

    assert weights_new[0] < weights_prev[0], \
        f"JKO did not reduce high-energy kernel: " \
        f"{weights_new[0]:.3f} vs {weights_prev[0]:.3f}"
```

Chapter 5

I/O and Persistence Tests

5.1 Atomic Snapshotting Test

```python
# tests/test_io/test_snapshotting.py
import pytest
import tempfile
import os
from Python.predictor import UniversalPredictor
from Python.config import PredictorConfig

def test_snapshot_save_load_integrity():
    """
    Test: Snapshot must preserve full state with checksum.
    """
    config = PredictorConfig()
    predictor1 = UniversalPredictor(config)

    for _ in range(50):
        obs = 100.0 + np.random.randn()
        predictor1.step_with_telemetry(obs, previous_target=obs)

    with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
        filepath = f.name

    try:
        predictor1.save_snapshot(filepath)

        predictor2 = UniversalPredictor(config)
        predictor2.load_snapshot(filepath)

        result1 = predictor1.step_with_telemetry(
            105.0, previous_target=105.0
        )
        result2 = predictor2.step_with_telemetry(
            105.0, previous_target=105.0
        )

        assert jnp.allclose(result1.weights, result2.weights, atol=1e-6), \
            "Weights mismatch after snapshot restore"

        assert jnp.allclose(
            result1.holder_exponent, result2.holder_exponent, atol=1e-6
        ), "Holder exponent mismatch"

    finally:
        os.unlink(filepath)

```

```python
def test_snapshot_corruption_detection():
    """
    Test: Corrupted snapshot must be rejected.
    """
    config = PredictorConfig()
    predictor1 = UniversalPredictor(config)

    with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
        filepath = f.name

    try:
        predictor1.save_snapshot(filepath)

        with open(filepath, 'rb+') as f:
            f.seek(100)
            f.write(b'\x00\x00\x00\x00')

        predictor2 = UniversalPredictor(config)

        with pytest.raises(ValueError, match="Checksum mismatch"):
            predictor2.load_snapshot(filepath)

    finally:
        os.unlink(filepath)

def test_snapshot_includes_telemetry():
    """
    Test: Snapshot must include kurtosis, DGM entropy, and flags.
    """
    import msgpack

    config = PredictorConfig()
    predictor = UniversalPredictor(config)

    for _ in range(300):
        obs = 100.0 + np.random.randn() * 5.0
        predictor.step_with_telemetry(obs, previous_target=obs)

    with tempfile.NamedTemporaryFile(delete=False, suffix='.msgpack') as f:
        filepath = f.name

    try:
        predictor.save_snapshot(filepath)

        with open(filepath, 'rb') as f:
            content = f.read()

        data_bytes = content[:-64]
        payload = msgpack.unpackb(data_bytes)

        assert 'telemetry' in payload, "Telemetry missing from snapshot"
        assert 'kurtosis' in payload['telemetry'], "Kurtosis not saved"
        assert 'dgm_entropy' in payload['telemetry'], "DGM entropy not saved"

        assert 'flags' in payload, "Flags missing from snapshot"
        assert 'degraded_inference' in payload['flags']
        assert 'emergency' in payload['flags']
        assert 'regime_change' in payload['flags']
        assert 'mode_collapse' in payload['flags']

    finally:
        os.unlink(filepath)
```

Chapter 6

Hardware Tests: CPU/GPU Parity

6.1 Numerical Consistency Test

```python
# tests/test_hardware/test_cpu_gpu_parity.py
import pytest
import jax
import jax.numpy as jnp
from Python.predictor import UniversalPredictor
from Python.config import PredictorConfig

@pytest.mark.parametrize("device", ["cpu", "gpu"])
def test_device_consistency(device):
    """
    Test: CPU and GPU must produce equivalent results.
    """
    if device == "gpu" and not jax.devices('gpu'):
        pytest.skip("GPU not available")

    with jax.default_device(jax.devices(device)[0]):
        config = PredictorConfig()
        predictor = UniversalPredictor(config)

        np.random.seed(555)
        data = np.random.randn(100) * 10.0 + 100.0

        results = []
        for obs in data:
            result = predictor.step_with_telemetry(obs, previous_target=obs)
            results.append({
                'prediction': float(result.predicted_next),
                'holder': float(result.holder_exponent),
                'weights': result.weights
            })

        return results

def test_cpu_gpu_parity():
    """
    Test: Compare CPU and GPU results.
    """
    if not jax.devices('gpu'):
        pytest.skip("GPU not available for parity test")

    results_cpu = test_device_consistency("cpu")
    results_gpu = test_device_consistency("gpu")

    for i, (cpu, gpu) in enumerate(zip(results_cpu, results_gpu)):
        assert jnp.allclose(
            cpu['weights'], gpu['weights'], rtol=1e-5, atol=1e-6
        ), f"Weights mismatch at step {i}"

        pred_diff = abs(cpu['prediction'] - gpu['prediction'])
        assert pred_diff < 1e-4, \
            f"Prediction mismatch at step {i}: {pred_diff:.2e}"
```

6.2 Hardware Parity with Quantization (FPGA Simulation)

```python
# tests/test_hardware/test_fixed_point_parity.py
import pytest
import jax.numpy as jnp
import numpy as np
from Python.predictor import UniversalPredictor

def quantize_to_fixed_point(x, int_bits=16, frac_bits=16):
    """Simulate fixed-point quantization Q16.16."""
    total_bits = int_bits + frac_bits
    max_val = (2 ** (total_bits - 1) - 1) / (2 ** frac_bits)
    min_val = -(2 ** (total_bits - 1)) / (2 ** frac_bits)

    x_clipped = jnp.clip(x, min_val, max_val)
    x_quantized = jnp.round(x_clipped * (2 ** frac_bits)) / (2 ** frac_bits)

    return x_quantized

def simulate_fpga_computation(prediction_float32):
    """Simulate FPGA pipeline: Float32 -> Q16.16 -> Q16.16."""
    pred_quantized_in = quantize_to_fixed_point(prediction_float32)
    intermediate = pred_quantized_in * 1.001
    pred_quantized_out = quantize_to_fixed_point(intermediate)
    return pred_quantized_out

def test_fpga_quantization_error():
    """
    Test: Q16.16 quantization must introduce <1% error.
    """
    config = UniversalPredictor.config
    predictor = UniversalPredictor(config)

    np.random.seed(777)
    data = 100.0 + np.random.randn(100) * 5.0

    predictions_float32 = []
    predictions_quantized = []

    for obs in data:
        result = predictor.step_with_telemetry(obs, previous_target=obs)
        pred_f32 = float(result.predicted_next)
        pred_quantized = float(simulate_fpga_computation(jnp.array(pred_f32)))

        predictions_float32.append(pred_f32)
        predictions_quantized.append(pred_quantized)

    preds_f32 = np.array(predictions_float32)
    preds_q = np.array(predictions_quantized)

    mask = np.abs(preds_f32) > 1e-3
    rel_error = np.abs(preds_f32[mask] - preds_q[mask]) / (np.abs(preds_f32[mask]) + 1e
    -6)

    max_rel_error = np.max(rel_error)
    mean_rel_error = np.mean(rel_error)

    assert max_rel_error < 0.01, \
        f"Max relative error too high: {max_rel_error:.2%}"

    assert mean_rel_error < 0.005, \
        f"Mean relative error too high: {mean_rel_error:.2%}"

def test_fpga_numerical_stability():
    """
    Test: Quantization accumulation remains bounded.
    """
    config = UniversalPredictor.config
    predictor_ref = UniversalPredictor(config)

    np.random.seed(888)
    data = 100.0 + np.random.randn(200) * 5.0

    predictions = []
    quantized_errors = []

    for i, obs in enumerate(data):
        result = predictor_ref.step_with_telemetry(obs, previous_target=obs)
        pred = float(result.predicted_next)
        pred_q = float(simulate_fpga_computation(jnp.array(pred)))

        predictions.append(pred)
        quantized_errors.append(abs(pred - pred_q))

    cumulative_error = np.cumsum(quantized_errors)
    final_cumulative = cumulative_error[-1]

    expected_max_cumulative = 200 * 1.5e-5 * 100

    assert final_cumulative < expected_max_cumulative * 10, \
        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.

```python
# tests/test_xla/test_no_host_device_sync.py
import pytest
import jax
import jax.numpy as jnp
from jax.core import Tracer
from Python.core.orchestrator import orchestrate_step
from Python.api.types import PredictorConfig, InternalState

def test_no_host_device_sync_in_orchestrator():
    """
    Ensure orchestration step returns unbacked DeviceArrays,
    not host floats.

    CRITICAL: Host-device synchronization blocks XLA dispatch,
    causing 100-500ms latency spikes and VRAM transfer overhead.
    """
    # Initialize configuration and state
    config = PredictorConfig()
    key = jax.random.PRNGKey(42)
    state = InternalState.initialize(config, key)

    # Mock signal input (on device)
    mock_signal = jnp.array(0.5)

    # Execute orchestration step
    new_state, prediction = orchestrate_step(mock_signal, state, config, key)

    # ASSERTION 1: Prediction must NOT be a Python float
    assert not isinstance(prediction, float), \
        "CRITICAL: Host-Device sync detected. " \
        "Prediction materialized as Python float instead of DeviceArray."

    # ASSERTION 2: Prediction must be JAX array type
    assert isinstance(prediction, (jnp.ndarray, jax.Array)), \
        f"Expected jax.Array, got {type(prediction)}"

    # ASSERTION 3: Array must have device attribute (not backed on host)
    assert hasattr(prediction, "device"), \
        "Prediction array must reside on XLA backend (CPU/GPU/TPU)."

    # ASSERTION 4: Verify device is not None (array is backed)
    assert prediction.device() is not None, \
        "Prediction array device is None (unbacked array)."

    # ASSERTION 5: State updates must also remain on device
    assert hasattr(new_state.dgm_entropy, "device"), \
        "State fields must remain on device for next iteration."

def test_telemetry_collection_lazy_evaluation():
    """
    Verify that telemetry fields use jax.lax.stop_gradient
    to prevent unnecessary computation during training.
    """
    from Python.io.telemetry import collect_telemetry

    # Mock state with tracked gradients
    state = create_mock_state_with_gradients()

    # Collect telemetry
    telemetry = collect_telemetry(state, config)

    # ASSERTION: Telemetry values must have stop_gradient applied
    # This prevents backprop through diagnostic metrics
    assert not hasattr(telemetry.kurtosis, "_trace"), \
        "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:

```python
# tests/test_xla/test_vmap_multi_tenant_parity.py
import pytest
import jax
import jax.numpy as jnp
from Python.core.orchestrator import orchestrate_step
from Python.api.types import PredictorConfig, InternalState

def test_vmap_multi_tenant_parity():
    """
    Verify that batched vmap execution is bit-exact
    to sequential execution.

    Multi-tenant deployments use vmap to process N clients
    in parallel. Any deviation between sequential and batched
    execution violates determinism guarantees.
    """
    batch_size = 128
    key = jax.random.PRNGKey(42)
    config = PredictorConfig()

    # Generate batch of signals and states
    signal_keys = jax.random.split(key, batch_size + 1)
    signals_batch = jax.random.normal(signal_keys[0], (batch_size, 100))

    # Initialize batched states
    state_keys = signal_keys[1:]
    states_batch = jax.vmap(
        lambda k: InternalState.initialize(config, k)
    )(state_keys)

    # SCENARIO 1: Sequential execution (baseline)
    seq_predictions = []
    seq_states = []

    for i in range(batch_size):
        new_state, prediction = orchestrate_step(
            signals_batch[i],
            states_batch[i],
            config,
            state_keys[i]
        )
        seq_predictions.append(prediction)
        seq_states.append(new_state)

    seq_predictions = jnp.stack(seq_predictions)

    # SCENARIO 2: Vectorized execution (production)
    vmap_orchestrate = jax.vmap(
        orchestrate_step,
        in_axes=(0, 0, None, 0)
    )

    batch_states, batch_predictions = vmap_orchestrate(
        signals_batch,
        states_batch,
        config,
        state_keys
    )

    # ASSERTION 1: Bit-exact prediction parity
    assert jnp.array_equal(seq_predictions, batch_predictions), \
        "XLA vmap compilation breaks mathematical parity. " \
        "Sequential and batched predictions must be bit-exact."

    # ASSERTION 2: State update parity
    for i in range(batch_size):
        assert jnp.array_equal(
            seq_states[i].dgm_entropy,
            batch_states.dgm_entropy[i]
        ), f"State divergence at index {i}: entropy mismatch"

        assert jnp.array_equal(
            seq_states[i].rho,
            batch_states.rho[i]
        ), f"State divergence at index {i}: rho weights mismatch"

    # ASSERTION 3: PRNG state advancement consistency
    # Next iteration must produce identical results
    next_signal = jnp.ones(batch_size)
    next_keys = jax.random.split(key, batch_size)

    _, next_pred_seq = orchestrate_step(
        next_signal[0], seq_states[0], config, next_keys[0]
    )
    _, next_pred_batch = vmap_orchestrate(
        next_signal, batch_states, config, next_keys
    )[1]

    assert jnp.array_equal(next_pred_seq, next_pred_batch[0]), \
        "PRNG state divergence detected after vmap execution."

def test_vmap_memory_efficiency():
    """
    Verify that vmap does not allocate N separate XLA buffers
    for identical config (memory amplification bug).
    """
    batch_size = 256
    config = PredictorConfig()

    # Single execution memory baseline
    baseline_memory = measure_peak_vram_usage(
        lambda: orchestrate_step(jnp.array(0.5), state, config, key)
    )

    # Batched execution memory
    batch_memory = measure_peak_vram_usage(
        lambda: jax.vmap(orchestrate_step, in_axes=(0, 0, None, 0))(
            signals_batch, states_batch, config, keys_batch
        )
    )

    # ASSERTION: Batch memory should scale sub-linearly
    # (not 256x single execution due to config sharing)
    expected_max_memory = baseline_memory * batch_size * 1.5

    assert batch_memory < expected_max_memory, \
        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.
```python
# tests/test_xla/test_load_shedding_jit_cache.py
import pytest
import time
import jax
from Python.api.warmup import warmup_kernel_d_load_shedding
from Python.kernels.kernel_d import kernel_d_predict
from Python.api.types import PredictorConfig

def test_load_shedding_warmup_no_recompilation():
    """
    Verify that swapping signature depths does not trigger
    JAX Cache Miss.

    Load shedding is a real-time emergency response to latency
    spikes. Triggering XLA recompilation (200ms) defeats the
    purpose of shedding (5ms target).
    """
    config = PredictorConfig(kernel_d_depth=5)
    key = jax.random.PRNGKey(42)
    signal = jax.random.normal(key, (100,))

    # PHASE 1: Warmup compiles M in {2, 3, 5}
    warmup_kernel_d_load_shedding(config, key)

    # PHASE 2: Baseline execution at M=5 (no load shedding)
    baseline_start = time.perf_counter()
    _ = kernel_d_predict(signal, key, config)
    baseline_time = time.perf_counter() - baseline_start

    # PHASE 3: Trigger load shedding M=5 -> M=2 (emergency mode)
    shedding_config = config.replace(kernel_d_depth=2)

    shedding_start = time.perf_counter()
    _ = kernel_d_predict(signal, key, shedding_config)
    shedding_time = time.perf_counter() - shedding_start

    # ASSERTION 1: Cached execution must be < 10ms
    # JIT compilation takes ~200ms. Cached execution < 5ms.
    assert shedding_time < 0.010, \
        f"CRITICAL: JIT Cache Miss during Load Shedding. " \
        f"Execution took {shedding_time*1000:.1f}ms (expected < 10ms). " \
        f"System will hang under stress."

    # ASSERTION 2: Shedding must not be slower than baseline
    # (Lower depth should be faster or equal)
    assert shedding_time <= baseline_time * 1.5, \
        f"Load shedding slower than baseline: " \
        f"{shedding_time/baseline_time:.2f}x"

    # PHASE 4: Verify cache hit for all depths
    for depth in [2, 3, 5]:
        test_config = config.replace(kernel_d_depth=depth)
        start = time.perf_counter()
        _ = kernel_d_predict(signal, key, test_config)
        exec_time = time.perf_counter() - start

        assert exec_time < 0.010, \
            f"Cache miss for depth={depth}: {exec_time*1000:.1f}ms"

def test_jit_cache_size_under_warmup():
    """
    Verify that warmup does not exhaust JIT cache memory limits.
    """
    import jax._src.xla_bridge as xb

    # Get initial cache size
    initial_cache = len(xb.get_backend().compile_cache())

    # Warmup all kernel variants
    warmup_kernel_d_load_shedding(config, key)

    # Get final cache size
    final_cache = len(xb.get_backend().compile_cache())

    # ASSERTION: Warmup should add exactly 3 entries (M=2,3,5)
    cache_growth = final_cache - initial_cache
    assert cache_growth == 3, \
        f"Unexpected cache growth: {cache_growth} entries " \
        f"(expected 3 for M in {{2,3,5}})"
```

Criterion 7.3. ***Acceptance Criteria:***

1. *Load shedding execution time:* $< 10ms$ *(cached) vs* $\sim 200ms$ *(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* ≤ 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
import pytest
import os
import tempfile
from pathlib import Path
from unittest.mock import patch, MagicMock
from Python.io.config_mutation import mutate_config
from Python.core.meta_optimizer import OptimizationResult

def test_atomic_toml_mutation():
    """
    Ensure config mutation uses temporary files and os.replace.

    POSIX Guarantee: os.replace() is atomic on Linux/macOS.
    Prevents partial writes visible to concurrent readers.
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        config_path = Path(tmpdir) / "config.toml"

        # Create initial config
        initial_params = {"cusum_k": 0.5, "learning_rate": 0.01}
        write_toml(config_path, initial_params)

        # Prepare mutation
        new_params = {"cusum_k": 0.8}
        validation_schema = {
            "cusum_k": {"range": [0.1, 2.0], "locked": False}
        }

        # MOCK os calls to verify protocol compliance
        with patch('os.replace') as mock_replace, \
             patch('os.fsync') as mock_fsync, \
             patch('os.open', return_value=3) as mock_open:

            # Trigger mutation
            mutate_config(new_params, config_path, validation_schema)

            # ASSERTION 1: Verify os.fsync was called
            # Ensures kernel buffer flush before atomic replace
            mock_fsync.assert_called_once()

            # ASSERTION 2: Verify os.replace was called with temp file
            assert mock_replace.call_count == 1
            args = mock_replace.call_args[0]

            # First arg must be temp file (config.toml.tmp)
            assert str(args[0]).endswith(".tmp"), \
                f"Expected temp file, got {args[0]}"

            # Second arg must be target file (config.toml)
            assert args[1] == config_path, \
                f"Expected {config_path}, got {args[1]}"

def test_concurrent_mutation_detection():
    """
    Verify that concurrent mutations are rejected
    (temp file already exists).
    """
    from Python.io.config_mutation import ConfigMutationError

    with tempfile.TemporaryDirectory() as tmpdir:
        config_path = Path(tmpdir) / "config.toml"
        tmp_path = config_path.with_suffix(".tmp")

        # Create initial config
        write_toml(config_path, {"param": 1.0})

        # Simulate concurrent mutation (temp file exists)
        tmp_path.touch()

        # ASSERTION: Mutation must fail with clear error
        with pytest.raises(ConfigMutationError,
                           match="Concurrent mutation detected"):
            mutate_config({"param": 2.0}, config_path, {})

```

```python
def test_audit_log_persistence():
    """
    Verify that mutation events are logged to io/mutations.log
    in JSON Lines format.
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        config_path = Path(tmpdir) / "config.toml"
        log_path = Path(tmpdir) / "mutations.log"

        write_toml(config_path, {"learning_rate": 0.01})

        # Perform mutation
        mutate_config(
            {"learning_rate": 0.015},
            config_path,
            {},
            audit_log_path=log_path
        )

        # ASSERTION: Audit log must exist and contain entry
        assert log_path.exists(), "Audit log not created"

        with open(log_path, 'r') as f:
            entries = [json.loads(line) for line in f]

        assert len(entries) == 1, "Expected 1 audit entry"

        entry = entries[0]
        assert entry["event"] == "MUTATION_APPLIED"
        assert "learning_rate" in entry["delta"]
        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)

```python
# tests/test_edge_cases/test_ttl_degraded_mode.py
import pytest
import jax.numpy as jnp
from Python.predictor import UniversalPredictorWithTelemetry
from Python.config import PredictorConfig

def test_degraded_mode_activation():
    """
    Test: Degraded mode activates when TTL exceeds limit.
    """
    config = PredictorConfig(staleness_ttl_ns=100_000_000)
    predictor = UniversalPredictorWithTelemetry(config)

    for _ in range(50):
        obs = 100.0 + np.random.randn()
        result = predictor.step_with_telemetry(obs, previous_target=obs)

    predictor.telemetry_logger.ttl_counter = 150

    obs = 100.0
    result = predictor.step_with_telemetry(obs, previous_target=obs)

    assert result.degraded_inference_mode, \
        "Degraded mode not activated despite TTL violation"

def test_degraded_mode_recovery_hysteresis():
    """
    Test: Recovery with hysteresis (0.8 * TTL_max).
    """
    config = PredictorConfig()
    predictor = UniversalPredictorWithTelemetry(config)

    predictor.telemetry_logger.ttl_counter = 150

    predictor.telemetry_logger.ttl_counter = 85

    result = predictor.step_with_telemetry(100.0, previous_target=100.0)
    assert result.degraded_inference_mode, \
        "Premature recovery (hysteresis not respected)"

    predictor.telemetry_logger.ttl_counter = 75

    result = predictor.step_with_telemetry(100.0, previous_target=100.0)
    assert not result.degraded_inference_mode, \
        "Recovery failed despite TTL below hysteresis threshold"
```

8.2 Extreme Kurtosis Test

```python
# tests/test_edge_cases/test_extreme_kurtosis.py
import pytest
import numpy as np
from Python.predictor import UniversalPredictorWithTelemetry
from Python.config import PredictorConfig

def test_extreme_kurtosis_detection():
    """
    Test: Kurtosis > 20 must generate critical alert.
    """
    config = PredictorConfig()
    predictor = UniversalPredictorWithTelemetry(config)

    from scipy.stats import t
    np.random.seed(666)
    extreme_data = t.rvs(df=2, size=500) * 20.0 + 100.0

    kurtosis_values = []

    for obs in extreme_data:
        result = predictor.step_with_telemetry(obs, previous_target=obs)
        kurtosis_values.append(float(result.kurtosis))

    final_kurtosis = kurtosis_values[-1]

    assert final_kurtosis > 15.0, \
        f"Extreme kurtosis not detected: kappa={final_kurtosis:.2f}"

    result = predictor.step_with_telemetry(
        extreme_data[-1], previous_target=extreme_data[-1]
    )

    h_adaptive = float(result.adaptive_threshold)
    h_fixed = config.cusum_h

    assert h_adaptive > 2.0 * h_fixed, \
        f"Adaptive threshold not sufficiently elevated: " \
        f"{h_adaptive:.2f} vs {h_fixed:.2f}"
```

Chapter 9

Walk-Forward Validation

```python
# tests/test_validation/test_walk_forward.py
import pytest
import numpy as np
from Python.validation import WalkForwardValidator
from Python.predictor import UniversalPredictor
from Python.config import PredictorConfig

def test_walk_forward_no_lookahead():
    """
    Test: Walk-forward must not use future information.
    """
    np.random.seed(777)
    T = 1000
    trend = np.linspace(100, 150, T)
    noise = np.random.randn(T) * 2.0
    data = trend + noise

    def model_factory(hp):
        config = PredictorConfig(
            epsilon=hp.get('epsilon', 1e-3),
            learning_rate=hp.get('tau', 0.1)
        )
        return UniversalPredictor(config)

    def metric_fn(preds, targets):
        return np.mean(np.abs(preds - targets))

    validator = WalkForwardValidator(
        model_factory=model_factory,
        metric_fn=metric_fn,
        window_size=252,
        horizon=1,
        max_memory=500
    )

    hyperparams = {'epsilon': 1e-2, 'tau': 0.05}

    mae = validator.run(data, hyperparams)

    data_range = np.max(data) - np.min(data)
    assert mae < 0.1 * data_range, \
        f"Walk-forward MAE too high: {mae:.2f}"

def test_walk_forward_regime_change():
    """
    Test: Performance under regime change.
    """
    np.random.seed(888)

    regime1 = np.linspace(100, 120, 400) + np.random.randn(400) * 1.0
    regime2 = np.linspace(120, 100, 400) + np.random.randn(400) * 1.0

    data = np.concatenate([regime1, regime2])

    def model_factory(hp):
        return UniversalPredictor(PredictorConfig())

    def metric_fn(preds, targets):
        return np.sqrt(np.mean((preds - targets)**2))

    validator = WalkForwardValidator(
        model_factory=model_factory,
        metric_fn=metric_fn,
        window_size=200,
        horizon=1
    )

    rmse = validator.run(data, {})

    assert rmse < 5.0, \
        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 `NaN`:*

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

3. *Run prediction on the poisoned series. If the model accesses future data, NaN propagates*
4. *Verify outputs:*

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

5. *Failure condition: if more than 0.1% of samples produce 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:

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

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

```python
# tests/test_causality/test_no_lookahead.py
import pytest
import jax.numpy as jnp
import numpy as np
from Python.predictor import UniversalPredictor
from Python.config import PredictorConfig

def test_predict_without_future_access():
    """
    Test: predict(t) must not access data with timestamp > t.
    """
    config = PredictorConfig()
    predictor = UniversalPredictor(config)

    np.random.seed(555)
    data = np.random.randn(100) * 10 + 100

    trap_position = 50
    trap_value = 1e6

    for i in range(trap_position):
        result = predictor.step_with_telemetry(
            data[i],
            previous_target=data[i]
        )

    buffer_ptr_before = id(predictor._state.signal_circular_buffer)
    internal_buffer_before = np.copy(predictor._state.signal_circular_buffer)

    result_at_t = predictor.step_with_telemetry(
        data[trap_position],
        previous_target=data[trap_position]
    )

    predictor._state.signal_circular_buffer = np.concatenate([
        predictor._state.signal_circular_buffer,
        jnp.array([trap_value])
    ])

    for i in range(trap_position + 1, trap_position + 6):
        if i < len(data):
            result_later = predictor.step_with_telemetry(
                data[i],
                previous_target=data[i]
            )

    predictor_clean = UniversalPredictor(config)
    for i in range(trap_position + 1):
        result_clean = predictor_clean.step_with_telemetry(
            data[i],
            previous_target=data[i]
        )

    pred_with_trap = float(result_at_t.predicted_next)
    pred_without_trap = float(result_clean.predicted_next)

    assert abs(pred_with_trap - pred_without_trap) < 1e-3, \
        f"Lookahead bias detected: pred_trap={pred_with_trap:.4f}, " \
        f"pred_clean={pred_without_trap:.4f}"

def test_causality_via_timestamps():
    """
    Test: Access timestamps should be monotonic.
    """
    config = PredictorConfig(wtmm_buffer_size=128)
    predictor = UniversalPredictor(config)

    original_buffer = predictor._state.signal_circular_buffer
    access_log = []

    class AccessTrackedBuffer:
        """Wrapper that logs access."""
        def __init__(self, buffer, log):
            self._buffer = buffer
            self._log = log

        def __getitem__(self, idx):
            import time
            timestamp = time.time_ns()
            self._log.append(('read', idx, timestamp))
            return self._buffer[idx]

        def __setitem__(self, idx, value):
            import time
            timestamp = time.time_ns()
            self._log.append(('write', idx, timestamp))
            self._buffer[idx] = value

        def __len__(self):
            return len(self._buffer)

    predictor._state.signal_circular_buffer = AccessTrackedBuffer(
        original_buffer, access_log
    )

    np.random.seed(666)
    data = np.random.randn(50) * 5 + 100

    for obs in data:
        predictor.step_with_telemetry(obs, previous_target=obs)

    read_indices = [idx for op, idx, _ in access_log if op == 'read']

    buffer_size = config.wtmm_buffer_size
    causal_violations = 0

    for i in range(1, len(read_indices)):
        curr_idx = read_indices[i] % buffer_size
        prev_idx = read_indices[i-1] % buffer_size

        if curr_idx < prev_idx and (prev_idx - curr_idx) > buffer_size // 2:
            causal_violations += 1

    assert causal_violations == 0, \
        f"Causal violations detected: {causal_violations} backward jumps"

def test_state_vector_does_not_leak_future():
    """
    Test: Sigma_t does not encode future information.
    """
    config = PredictorConfig()

    predictor1 = UniversalPredictor(config)
    data_short = np.random.randn(50) * 5 + 100

    for obs in data_short:
        result1 = predictor1.step_with_telemetry(obs, previous_target=obs)

    state1_weights = np.copy(predictor1._state.weights)
    state1_cusum = np.copy(predictor1._state.cusum_acum if hasattr(predictor1._state, '
    cusum_acum') else [])

    predictor2 = UniversalPredictor(config)
    np.random.seed(np.random.RandomState(42).randint(2**32))
    data_long = np.random.randn(100) * 5 + 100

    for i in range(50):
        result2 = predictor2.step_with_telemetry(data_long[i], previous_target=data_long[
    i])

    state2_weights = np.copy(predictor2._state.weights)
    state2_cusum = np.copy(predictor2._state.cusum_acum if hasattr(predictor2._state, '
    cusum_acum') else [])

    weights_diff = np.max(np.abs(state1_weights - state2_weights))

    assert weights_diff < 0.05, \
        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%
Total			**91%**

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.

```bash
#!/bin/bash
# Pre-pytest environment validation

# Extract versions from requirements.txt (handles environment markers)
# Format: "jax==0.4.38; sys_platform == 'darwin' and platform_machine == 'x86_64'"
EXPECTED_JAX=$(grep "^jax==" ../requirements.txt | head -1 | cut -d'=' -f3 | cut -d';' -
    f1)
EXPECTED_EQUINOX=$(grep "^equinox==" ../requirements.txt | cut -d'=' -f3)
EXPECTED_DIFFRAX=$(grep "^diffrax==" ../requirements.txt | cut -d'=' -f3)

ACTUAL_JAX=$(python -c "import jax; print(jax.__version__)")
```

```bash
ACTUAL_EQUINOX=$(python -c "import equinox; print(equinox.__version__)")
ACTUAL_DIFFRAX=$(python -c "import diffrax; print(diffrax.__version__)")

if [ "$EXPECTED_JAX" != "$ACTUAL_JAX" ]; then
    echo "ERROR: JAX mismatch - Expected $EXPECTED_JAX, got $ACTUAL_JAX"
    exit 1
fi

if [ "$EXPECTED_EQUINOX" != "$ACTUAL_EQUINOX" ]; then
    echo "ERROR: Equinox mismatch - Expected $EXPECTED_EQUINOX, got $ACTUAL_EQUINOX"
    exit 1
fi

if [ "$EXPECTED_DIFFRAX" != "$ACTUAL_DIFFRAX" ]; then
    echo "ERROR: Diffrax mismatch - Expected $EXPECTED_DIFFRAX, got $ACTUAL_DIFFRAX"
    exit 1
fi

echo "  Environment validation OK - Proceed with pytest"
```

Listing 11.1: Pre-Test Environment Validation

11.2.2 Execution Commands

```bash
# Run all tests with coverage report
pytest tests/ -v --cov=Python --cov-report=html

# Run only fast tests (exclude GPU and optimization)
pytest tests/ -v -m "not slow"

# Run GPU parity tests (if available)
pytest tests/test_hardware/ -v -k gpu

# Parallel tests (4 workers)
pytest tests/ -n 4 --dist loadscope

# Generate XML report for CI/CD
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