```
        """Test scoring signals have all required fields."""
        signals = signal_registry.get_scoring_signals("Q001")

        # Check required fields
        assert "Q001" in signals.question_modalities
        assert len(signals.modality_configs) > 0
        assert len(signals.quality_levels) == 4  # Must be exactly 4
        assert signals.source_hash is not None

        # Check quality levels are ordered
        min_scores = [lvl.min_score for lvl in signals.quality_levels]
        assert min_scores == sorted(min_scores, reverse=True)


# ============================================================================
# TEST 5: OBSERVABILITY & METRICS
# ============================================================================


class TestObservability:
    """Test observability and metrics collection."""

    def test_metrics_collection(self, signal_registry):
        """Test that metrics are collected."""
        # Access some signals
        signal_registry.get_chunking_signals()
        signal_registry.get_micro_answering_signals("Q001")

        # Get metrics
        metrics = signal_registry.get_metrics()

        # Check metrics structure
        assert "cache_hits" in metrics
        assert "cache_misses" in metrics
        assert "hit_rate" in metrics
        assert "signal_loads" in metrics

        # Verify counts
        assert metrics["cache_hits"] >= 0
        assert metrics["cache_misses"] >= 0
        assert 0.0 <= metrics["hit_rate"] <= 1.0

    def test_cache_hit_rate_improvement(self, signal_registry):
        """Test that cache hit rate improves with repeated access."""
        # Clear cache
        signal_registry.clear_cache()

        # First access (cold cache)
        for i in range(1, 6):
            try:
                signal_registry.get_micro_answering_signals(f"Q{i:03d}")
            except (ValueError, KeyError):
                pass

        metrics_cold = signal_registry.get_metrics()
        cold_hit_rate = metrics_cold["hit_rate"]

        # Second access (warm cache)
        for i in range(1, 6):
            try:
                signal_registry.get_micro_answering_signals(f"Q{i:03d}")
            except (ValueError, KeyError):
                pass

        metrics_warm = signal_registry.get_metrics()
        warm_hit_rate = metrics_warm["hit_rate"]

        print(f"\n=== CACHE HIT RATE PROGRESSION ===")
        print(f"Cold cache hit rate: {cold_hit_rate:.1%}")
        print(f"Warm cache hit rate: {warm_hit_rate:.1%}")
        print(f"Improvement: {(warm_hit_rate - cold_hit_rate):.1%}")

        # Warm cache should have higher hit rate
        assert warm_hit_rate >= cold_hit_rate


# ============================================================================
# TEST 6: INTEGRATION TEST
# ============================================================================


class TestIntegration:
    """End-to-end integration tests."""

    def test_full_signal_pipeline(self, signal_registry):
        """Test complete signal retrieval pipeline."""
        question_id = "Q001"

        # 1. Get chunking signals
        chunking = signal_registry.get_chunking_signals()
        assert chunking is not None

        # 2. Get micro answering signals
        answering = signal_registry.get_micro_answering_signals(question_id)
        assert answering is not None

        # 3. Get validation signals
        validation = signal_registry.get_validation_signals(question_id)
        assert validation is not None

        # 4. Get scoring signals
        scoring = signal_registry.get_scoring_signals(question_id)
        assert scoring is not None

        # All signal packs should have same source hash (content-addressed)
        assert chunking.source_hash == answering.source_hash
        assert answering.source_hash == validation.source_hash
        assert validation.source_hash == scoring.source_hash

    def test_signal_consistency_across_questions(self, signal_registry):
        """Test that signals are consistent for same policy area."""
        # Get signals for two questions in same policy area
        signals_q001 = signal_registry.get_micro_answering_signals("Q001")
        signals_q002 = signal_registry.get_micro_answering_signals("Q002")

        # Should have same policy area indicators if same PA
        # (both Q001 and Q002 are PA01 - gender)
        pa_q001 = list(signals_q001.indicators_by_pa.keys())
        pa_q002 = list(signals_q002.indicators_by_pa.keys())

        # Both should reference PA01
        assert "PA01" in pa_q001 or len(pa_q001) == 0
        assert "PA01" in pa_q002 or len(pa_q002) == 0


# ============================================================================
# CONTRAFACTUAL SUMMARY TEST
# ============================================================================


class TestContrafactualSummary:
    """Generate comprehensive contrafactual analysis summary."""

    def test_generate_contrafactual_report(self, signal_registry):
        """Generate full contrafactual comparison report."""
        report = {
```

```
    "pattern_precision": {},
    "performance": {},
    "type_safety": {},
    "cache_efficiency": {},
}

# Pattern precision
tester = TestPatternMatchPrecision()
baseline_indicators = tester.test_indicator_extraction_without_signals()
signal_indicators = tester.test_indicator_extraction_with_signals(
    signal_registry
)

report["pattern_precision"] = {
    "baseline": baseline_indicators,
    "with_signals": signal_indicators,
    "improvement_pct": (
        ((signal_indicators - baseline_indicators) / baseline_indicators * 100)
        if baseline_indicators > 0
        else 0
    ),
}

# Performance
perf_tester = TestPerformanceBenchmark()
cold_time = perf_tester.test_signal_loading_cold_cache(signal_registry)
warm_time = perf_tester.test_signal_loading_warm_cache(signal_registry)

report["performance"] = {
    "cold_cache_ms": cold_time * 1000,
    "warm_cache_ms": warm_time * 1000,
    "speedup": (cold_time / warm_time) if warm_time > 0 else 1.0,
}

# Cache efficiency
metrics = signal_registry.get_metrics()
report["cache_efficiency"] = {
    "hit_rate": metrics["hit_rate"],
    "cache_hits": metrics["cache_hits"],
    "cache_misses": metrics["cache_misses"],
}

# Print comprehensive report
print("\n" + "=" * 70)
print("CONTRAFACTUAL ANALYSIS SUMMARY")
print("=" * 70)

print("\n1. PATTERN PRECISION")
print(f"   Baseline:      {report['pattern_precision']['baseline']} matches")
print(f"   With Signals:  {report['pattern_precision']['with_signals']} matches")
print(
    f"   Improvement:   {report['pattern_precision']['improvement_pct']:+.1f}%"
)

print("\n2. PERFORMANCE")
print(f"   Cold Cache:    {report['performance']['cold_cache_ms']:.2f}ms")
print(f"   Warm Cache:    {report['performance']['warm_cache_ms']:.2f}ms")
print(f"   Speedup:       {report['performance']['speedup']:.1f}x")

print("\n3. CACHE EFFICIENCY")
print(f"   Hit Rate:      {report['cache_efficiency']['hit_rate']:.1%}")
print(f"   Cache Hits:    {report['cache_efficiency']['cache_hits']}")
print(f"   Cache Misses:  {report['cache_efficiency']['cache_misses']}")

print("\n" + "=" * 70)

# Assertions
assert report["cache_efficiency"]["hit_rate"] >= 0.0
```

```
        assert report["performance"]["speedup"] >= 1.0 or report["performance"][
            "warm_cache_ms"
        ] == 0


# ============================================================================
# PYTEST CONFIGURATION
# ============================================================================


def pytest_configure(config):
    """Configure pytest with custom markers."""
    config.addinivalue_line(
        "markers", "contrafactual: mark test as contrafactual analysis"
    )
    config.addinivalue_line("markers", "benchmark: mark test as performance benchmark")
    config.addinivalue_line("markers", "integration: mark test as integration test")



if __name__ == "__main__":
    pytest.main([__file__, "-v", "-s"])

===== FILE: tests/test_signal_pa_dimension_coverage.py =====
"""
Test for signal policy area and dimension coverage.
"""

import json
from pathlib import Path
import pytest
from typing import Any, Dict, List

# Add src to python path for imports (if needed for fixtures, etc.)
import sys
sys.path.append(str(Path(__file__).parent.parent / "src"))

PROJECT_ROOT = Path(__file__).parent.parent.resolve()
SIGNAL_AUDIT_MANIFEST_PATH = PROJECT_ROOT / "artifacts" / "audit" /
"signal_audit_manifest.json"

@pytest.fixture(scope="module")
def signal_audit_metrics() -> Dict[str, Any]:
    """
    Pytest fixture to load the signal audit manifest metrics.
    """
    if not SIGNAL_AUDIT_MANIFEST_PATH.exists():
        pytest.fail(f"Signal audit manifest not found at {SIGNAL_AUDIT_MANIFEST_PATH}")

    manifest = json.loads(SIGNAL_AUDIT_MANIFEST_PATH.read_text(encoding="utf-8"))
    return manifest.get("metrics", {})

def test_pa_signal_coverage(signal_audit_metrics: Dict[str, Any]):
    """
    Tests that policy area signal coverage is greater than or equal to 90%.
    """
    pa_coverage_percentage = signal_audit_metrics.get("pa_coverage_percentage", 0.0)
    expected_min_coverage = 90.0

    assert pa_coverage_percentage >= expected_min_coverage, (
        f"Policy Area signal coverage ({pa_coverage_percentage:.2f}%) "
        f"is below the required minimum of {expected_min_coverage}%."
    )

def test_dimensions_with_signals(signal_audit_metrics: Dict[str, Any]):
    """
    Tests that at least some dimensions have signals defined.
    (This is a basic check; more detailed validation would require deeper logic).
    """
    dimensions_with_signals = signal_audit_metrics.get("dimensions_with_signals", 0)

    assert dimensions_with_signals > 0, (
        "No dimensions were found to have signals defined. "
        "Expected at least one dimension to be covered by signals."
    )

===== FILE: tests/test_signals.py =====
"""Tests for Cross-Cut Signal Channel Implementation.

This module tests:
- SignalPack creation, validation, and hashing
- SignalRegistry (TTL, LRU, caching)
- InMemorySignalSource
- SignalClient (memory:// and HTTP modes)
- Circuit breaker behavior
- HTTP status code mapping
- ETag support
"""

from __future__ import annotations

import time
from datetime import datetime, timezone
from typing import Any
from unittest.mock import Mock, patch

import pytest
from pydantic import ValidationError

from saaaaaa.core.orchestrator.signals import (
    SignalPack,
    SignalRegistry,
    SignalClient,
    InMemorySignalSource,
    CircuitBreakerError,
    SignalUnavailableError,
    PolicyArea,
    create_default_signal_pack,
)


# ============================================================================
# SignalPack Tests
# ============================================================================


def test_signal_pack_creation() -> None:
    """Test basic SignalPack creation."""
    pack = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["pattern1", "pattern2"],
        indicators=["indicator1"],
        regex=[r"\d+"],
        verbs=["implement", "execute"],
        entities=["government", "budget"],
        thresholds={"min_confidence": 0.85, "min_evidence": 0.75},
    )

    assert pack.version == "1.0.0"
    assert pack.policy_area == "fiscal"
    assert len(pack.patterns) == 2
    assert len(pack.indicators) == 1
    assert pack.thresholds["min_confidence"] == 0.85


def test_signal_pack_frozen() -> None:
    """Test that SignalPack is immutable."""
    pack = SignalPack(version="1.0.0", policy_area="fiscal")

    with pytest.raises(ValidationError):
        pack.version = "2.0.0"  # type: ignore


def test_signal_pack_version_validation() -> None:
    """Test semantic version validation."""
    # Valid versions
    SignalPack(version="1.0.0", policy_area="fiscal")
    SignalPack(version="10.20.30", policy_area="salud")

    # Invalid versions
    with pytest.raises(ValidationError):
        SignalPack(version="1.0", policy_area="fiscal")

    with pytest.raises(ValidationError):
        SignalPack(version="v1.0.0", policy_area="fiscal")

    with pytest.raises(ValidationError):
        SignalPack(version="1.0.0-beta", policy_area="fiscal")


def test_signal_pack_threshold_validation() -> None:
    """Test threshold value validation."""
    # Valid thresholds
    SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        thresholds={"min": 0.0, "max": 1.0, "mid": 0.5},
    )

    # Invalid threshold (out of range)
    with pytest.raises(ValidationError):
        SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            thresholds={"min": 1.5},
        )

    with pytest.raises(ValidationError):
        SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            thresholds={"min": -0.1},
        )


def test_signal_pack_compute_hash() -> None:
    """Test deterministic hash computation."""
    pack1 = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["a", "b"],
        indicators=["i1"],
    )

    pack2 = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["a", "b"],
        indicators=["i1"],
    )

    # Same content → same hash
    assert pack1.compute_hash() == pack2.compute_hash()
```

```python
    # Different content → different hash
    pack3 = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["a", "c"],
        indicators=["i1"],
    )
    assert pack1.compute_hash() != pack3.compute_hash()


def test_signal_pack_hash_stability() -> None:
    """Test hash stability (property test for BLAKE3)."""
    pack = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["pattern1", "pattern2"],
    )

    # Hash should be stable across multiple calls
    hash1 = pack.compute_hash()
    hash2 = pack.compute_hash()
    hash3 = pack.compute_hash()

    assert hash1 == hash2 == hash3


def test_signal_pack_is_valid() -> None:
    """Test validity window checking."""
    now = datetime.now(timezone.utc)

    # Pack is valid now
    pack1 = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        valid_from=now.isoformat(),
        valid_to="",
    )
    assert pack1.is_valid(now)

    # Pack with expired valid_to
    pack2 = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        valid_from="2020-01-01T00:00:00+00:00",
        valid_to="2021-01-01T00:00:00+00:00",
    )
    assert not pack2.is_valid(now)


def test_signal_pack_get_keys_used() -> None:
    """Test keys_used extraction."""
    pack = SignalPack(
        version="1.0.0",
        policy_area="fiscal",
        patterns=["p1"],
        indicators=["i1"],
        regex=[],
        verbs=[],
        entities=[],
        thresholds={},
    )

    keys = pack.get_keys_used()
    assert "patterns" in keys
    assert "indicators" in keys
    assert "regex" not in keys  # Empty list
    assert "thresholds" not in keys  # Empty dict
```

```
# ==========================================================================
# SignalRegistry Tests
# ==========================================================================


def test_signal_registry_initialization() -> None:
    """Test registry initialization."""
    registry = SignalRegistry(max_size=10, default_ttl_s=60)

    metrics = registry.get_metrics()
    assert metrics["size"] == 0
    assert metrics["capacity"] == 10
    assert metrics["hit_rate"] == 0.0


def test_signal_registry_put_get() -> None:
    """Test basic put/get operations."""
    registry = SignalRegistry(max_size=10, default_ttl_s=60)

    pack = SignalPack(version="1.0.0", policy_area="fiscal")
    registry.put("fiscal", pack)

    retrieved = registry.get("fiscal")
    assert retrieved is not None
    assert retrieved.version == "1.0.0"
    assert retrieved.policy_area == "fiscal"


def test_signal_registry_ttl_expiration() -> None:
    """Test TTL-based expiration."""
    registry = SignalRegistry(max_size=10, default_ttl_s=1)

    pack = SignalPack(version="1.0.0", policy_area="fiscal", ttl_s=1)
    registry.put("fiscal", pack)

    # Should be available immediately
    assert registry.get("fiscal") is not None

    # Wait for expiration
    time.sleep(1.5)

    # Should be expired
    assert registry.get("fiscal") is None


def test_signal_registry_lru_eviction() -> None:
    """Test LRU eviction when capacity exceeded."""
    registry = SignalRegistry(max_size=3, default_ttl_s=3600)

    # Fill to capacity
    pack1 = SignalPack(version="1.0.0", policy_area="fiscal")
    pack2 = SignalPack(version="1.0.0", policy_area="salud")
    pack3 = SignalPack(version="1.0.0", policy_area="ambiente")

    registry.put("fiscal", pack1)
    registry.put("salud", pack2)
    registry.put("ambiente", pack3)

    # Add one more → should evict oldest (fiscal)
    pack4 = SignalPack(version="1.0.0", policy_area="energía")
    registry.put("energía", pack4)

    assert registry.get("fiscal") is None  # Evicted
    assert registry.get("salud") is not None
    assert registry.get("ambiente") is not None
    assert registry.get("energía") is not None
```

```
def test_signal_registry_hit_rate() -> None:
    """Test hit rate calculation."""
    registry = SignalRegistry(max_size=10, default_ttl_s=3600)

    pack = SignalPack(version="1.0.0", policy_area="fiscal")
    registry.put("fiscal", pack)

    # 3 hits, 2 misses
    registry.get("fiscal")
    registry.get("fiscal")
    registry.get("fiscal")
    registry.get("salud")
    registry.get("ambiente")

    metrics = registry.get_metrics()
    assert metrics["hits"] == 3
    assert metrics["misses"] == 2
    assert metrics["hit_rate"] == 0.6  # 3/5


def test_signal_registry_clear() -> None:
    """Test clearing registry."""
    registry = SignalRegistry(max_size=10, default_ttl_s=3600)

    pack1 = SignalPack(version="1.0.0", policy_area="fiscal")
    pack2 = SignalPack(version="1.0.0", policy_area="salud")
    registry.put("fiscal", pack1)
    registry.put("salud", pack2)

    assert registry.get_metrics()["size"] == 2

    registry.clear()

    assert registry.get_metrics()["size"] == 0
    assert registry.get("fiscal") is None
    assert registry.get("salud") is None


# ============================================================================
# InMemorySignalSource Tests
# ============================================================================


def test_in_memory_signal_source_register_get() -> None:
    """Test in-memory signal source registration and retrieval."""
    source = InMemorySignalSource()

    pack = SignalPack(version="1.0.0", policy_area="fiscal")
    source.register("fiscal", pack)

    retrieved = source.get("fiscal")
    assert retrieved is not None
    assert retrieved.version == "1.0.0"


def test_in_memory_signal_source_miss() -> None:
    """Test in-memory signal source miss."""
    source = InMemorySignalSource()

    result = source.get("nonexistent")
    assert result is None


# ============================================================================
# SignalClient (memory://) Tests
# ============================================================================
def test_signal_client_memory_mode() -> None:
    """Test signal client in memory:// mode."""
    client = SignalClient(base_url="memory://")

    # Register signal
    pack = SignalPack(version="1.0.0", policy_area="fiscal")
    client.register_memory_signal("fiscal", pack)

    # Fetch signal
    retrieved = client.fetch_signal_pack("fiscal")
    assert retrieved is not None
    assert retrieved.version == "1.0.0"


def test_signal_client_memory_mode_miss() -> None:
    """Test signal client memory:// miss."""
    client = SignalClient(base_url="memory://")

    result = client.fetch_signal_pack("nonexistent")
    assert result is None


def test_signal_client_memory_mode_metrics() -> None:
    """Test signal client metrics in memory:// mode."""
    client = SignalClient(base_url="memory://")

    metrics = client.get_metrics()
    assert metrics["transport"] == "memory"
    assert metrics["circuit_open"] is False


def test_signal_client_register_memory_signal_http_mode_error() -> None:
    """Test that register_memory_signal raises error in HTTP mode."""
    client = SignalClient(
        base_url="http://localhost:8000",
        enable_http_signals=True,
    )

    pack = SignalPack(version="1.0.0", policy_area="fiscal")

    with pytest.raises(ValueError, match="memory:// mode"):
        client.register_memory_signal("fiscal", pack)


# ============================================================================
# SignalClient (HTTP) Tests with Mock Transport
# ============================================================================


def test_signal_client_http_200_success() -> None:
    """Test HTTP 200 OK response."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    # Use httpx MockTransport
    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(
            200,
            json={"version": "1.0.0", "policy_area": "fiscal"},
            headers={"ETag": "abc123"},
        )

    mock_transport = httpx.MockTransport(handler)

    # Monkey-patch httpx.get to use our transport
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        pack = client.fetch_signal_pack("fiscal")

        assert pack is not None
        assert pack.version == "1.0.0"
        assert pack.policy_area == "fiscal"
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_304_not_modified() -> None:
    """Test HTTP 304 Not Modified response."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(304)

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        result = client.fetch_signal_pack("fiscal", etag="abc123")

        assert result is None
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_401_unauthorized() -> None:
    """Test HTTP 401 Unauthorized response."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(401, text="Unauthorized")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get
    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        with pytest.raises(SignalUnavailableError) as exc_info:
            client.fetch_signal_pack("fiscal")

        assert exc_info.value.status_code == 401
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_429_rate_limit() -> None:
    """Test HTTP 429 Too Many Requests response."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(429, text="Too Many Requests")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        with pytest.raises(SignalUnavailableError) as exc_info:
            client.fetch_signal_pack("fiscal")

        assert exc_info.value.status_code == 429
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_500_server_error() -> None:
    """Test HTTP 500 Server Error response."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(500, text="Internal Server Error")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        with pytest.raises(SignalUnavailableError) as exc_info:
            client.fetch_signal_pack("fiscal")

        assert exc_info.value.status_code == 500
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_timeout() -> None:
    """Test HTTP timeout."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        raise httpx.TimeoutException("Timeout")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
            timeout_s=1.0,
        )

        with pytest.raises(SignalUnavailableError) as exc_info:
            client.fetch_signal_pack("fiscal")

        assert "timeout" in str(exc_info.value).lower()
    finally:
        httpx.get = original_get  # type: ignore


def test_signal_client_http_response_size_limit() -> None:
    """Test response size limit enforcement."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    # Create response exceeding 1.5 MB
    large_content = b"x" * (SignalClient.MAX_RESPONSE_SIZE_BYTES + 1)

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(200, content=large_content)

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get
    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
        )

        with pytest.raises(SignalUnavailableError, match="exceeds maximum"):
            client.fetch_signal_pack("fiscal")
    finally:
        httpx.get = original_get  # type: ignore


# ============================================================================
# Circuit Breaker Tests
# ============================================================================


def test_circuit_breaker_opens_after_threshold() -> None:
    """Test circuit breaker opens after threshold failures."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(500, text="Error")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
            circuit_breaker_threshold=3,
            max_retries=1,  # Minimize retries for faster test
        )

        # First 3 failures should open circuit (threshold=3)
        for i in range(3):
            try:
                client.fetch_signal_pack("fiscal")
            except (SignalUnavailableError, CircuitBreakerError):
                pass  # Expected

        # Circuit should be open now
        assert client._circuit_open is True

        # Next call should raise CircuitBreakerError immediately
        with pytest.raises(CircuitBreakerError):
            client.fetch_signal_pack("fiscal")
    finally:
        httpx.get = original_get  # type: ignore


def test_circuit_breaker_closes_after_cooldown() -> None:
    """Test circuit breaker closes after cooldown period."""
    try:
        import httpx
    except ImportError:
        pytest.skip("httpx not installed")

    def handler(request: httpx.Request) -> httpx.Response:
        return httpx.Response(500, text="Error")

    mock_transport = httpx.MockTransport(handler)
    original_get = httpx.get

    def mock_get(url: str, **kwargs: Any) -> httpx.Response:
        client = httpx.Client(transport=mock_transport)
        return client.get(url, **kwargs)

    httpx.get = mock_get  # type: ignore

    try:
        client = SignalClient(
            base_url="http://localhost:8000",
            enable_http_signals=True,
            circuit_breaker_threshold=2,
            circuit_breaker_cooldown_s=1.0,
            max_retries=1,  # Minimize retries
        )

        # Trigger circuit breaker
        for i in range(2):
            try:
                client.fetch_signal_pack("fiscal")
            except (SignalUnavailableError, CircuitBreakerError):
                pass  # Expected

        assert client._circuit_open is True

        # Wait for cooldown
        time.sleep(1.5)

        # Circuit should try to close and fail again
        try:
            client.fetch_signal_pack("fiscal")
        except (SignalUnavailableError, CircuitBreakerError):
            pass  # Expected - still failing but circuit tried to close

        # Circuit should have attempted to close
        # (failure_count would be reset to 0 before the attempt)
    finally:
        httpx.get = original_get  # type: ignore


# ============================================================================
# Helper Function Tests
# ============================================================================


def test_create_default_signal_pack() -> None:
    """Test default signal pack creation."""
    pack = create_default_signal_pack("fiscal")

    assert pack.version == "0.0.0"
    assert pack.policy_area == "fiscal"
    assert pack.thresholds["min_confidence"] == 0.9
    assert pack.metadata["mode"] == "conservative_fallback"


# ============================================================================
# Integration Tests
# ==========================================================================


def test_signal_client_with_registry_integration() -> None:
    """Test integration of SignalClient with SignalRegistry."""
    client = SignalClient(base_url="memory://")
    registry = SignalRegistry(max_size=10, default_ttl_s=3600)

    # Register signal in client
    pack = SignalPack(version="1.0.0", policy_area="fiscal", patterns=["test"])
    client.register_memory_signal("fiscal", pack)

    # Fetch and store in registry
    fetched = client.fetch_signal_pack("fiscal")
    assert fetched is not None

    registry.put("fiscal", fetched)

    # Retrieve from registry
    cached = registry.get("fiscal")
    assert cached is not None
    assert cached.version == "1.0.0"


def test_http_url_without_enable_flag_falls_back_to_memory() -> None:
    """Test that HTTP URL without enable_http_signals flag falls back to memory://."""
    client = SignalClient(
        base_url="http://localhost:8000",
        enable_http_signals=False,
    )

    metrics = client.get_metrics()
    assert metrics["transport"] == "memory"


def test_invalid_url_scheme_raises_error() -> None:
    """Test that invalid URL scheme raises error."""
    with pytest.raises(ValueError, match="Invalid base_url scheme"):
        SignalClient(base_url="ftp://localhost")

===== FILE: tests/test_signature_validation.py =====
"""
Tests for signature validation and defensive programming fixes
"""

import sys
from pathlib import Path

# Add project root to path

def test_defensive_function_with_extra_kwargs():
    """Test defensive function that accepts unexpected kwargs"""

    # Mock defensive function similar to _is_likely_header
    def defensive_function(required_param, **kwargs):
        """Defensive implementation that accepts extra kwargs"""
        if kwargs:
            print(f"Warning: Received unexpected kwargs: {list(kwargs.keys())}")

        return f"Processed: {required_param}"

    # Test case 1: Normal usage
    result = defensive_function("test_value")
    assert result == "Processed: test_value"
    print("✓ Normal usage works")

    # Test case 2: With unexpected kwargs (this would have failed before)
    result = defensive_function("test_value", pdf_path="/some/path.pdf", extra="value")
    assert result == "Processed: test_value"
    print("✓ Accepts unexpected kwargs without crashing")

    print("\n✓ test_defensive_function_with_extra_kwargs PASSED")

def test_defensive_function_with_optional_param():
    """Test defensive function that handles missing required parameter"""

    # Mock defensive function similar to _analyze_causal_dimensions
    def defensive_function(text, sentences=None):
        """Defensive implementation that handles missing sentences"""
        if sentences is None:
            print("Warning: Missing 'sentences' parameter, using fallback")
            # Fallback: simple split
            sentences = text.split('. ')

        return {
            "sentence_count": len(sentences),
            "text_length": len(text)
        }

    # Test case 1: With parameter provided
    result = defensive_function(
        "Text content.",
        sentences=["Text content."]
    )
    assert result["sentence_count"] == 1
    print("✓ Works with sentences provided")

    # Test case 2: Without parameter (this would have failed before)
    result = defensive_function("Sentence one. Sentence two.")
    assert result["sentence_count"] == 2
    print("✓ Works without optional parameter")

    print("\n✓ test_defensive_function_with_optional_param PASSED")

def test_defensive_class_init_with_extra_kwargs():
    """Test defensive class __init__ that accepts unexpected kwargs"""

    # Mock class with defensive __init__ similar to BayesianMechanismInference
    class DefensiveClass:
        def __init__(self, config, nlp_model, **kwargs):
            """Defensive __init__ that accepts extra kwargs"""
            if kwargs:
                print(f"Warning: Received unexpected kwargs: {list(kwargs.keys())}")

            self.config = config
            self.nlp_model = nlp_model

    # Test case 1: Normal usage
    obj1 = DefensiveClass(
        config="mock_config",
        nlp_model="mock_nlp"
    )
    assert obj1.config == "mock_config"
    print("✓ Normal instantiation works")

    # Test case 2: With unexpected kwargs (would have failed before)
    obj2 = DefensiveClass(
        config="mock_config",
        nlp_model="mock_nlp",
        causal_hierarchy={"some": "data"},
        unexpected_param="value"
    )
    assert obj2.config == "mock_config"
    print("✓ Accepts unexpected kwargs without crashing")

    print("\n✓ test_defensive_class_init_with_extra_kwargs PASSED")
```

```
def test_signature_validator_basic_functionality():
    """Test basic signature validation functionality"""

    from saaaaaa.utils.signature_validator import validate_call_signature

    def sample_function(arg1: str, arg2: int):
        return f"{arg1}: {arg2}"

    # Valid call
    assert validate_call_signature(sample_function, "test", 123)
    print("✓ Valid call detected correctly")

    # Invalid call - missing argument
    assert not validate_call_signature(sample_function, "test")
    print("✓ Missing argument detected")

    # Invalid call - too many positional arguments
    assert not validate_call_signature(sample_function, "test", 123, "extra")
    print("✓ Too many arguments detected")

    print("\n✓ test_signature_validator_basic_functionality PASSED")

def test_validate_signature_decorator():
    """Test the validate_signature decorator"""

    from saaaaaa.utils.signature_validator import validate_signature

    @validate_signature(enforce=False, track=False)
    def decorated_function(param1: str, param2: int) -> str:
        return f"{param1}-{param2}"

    # Test normal call
    result = decorated_function("test", 123)
    assert result == "test-123"
    print("✓ Decorated function works normally")

    # Test call with kwargs
    result = decorated_function(param1="test", param2=456)
    assert result == "test-456"
    print("✓ Decorated function works with kwargs")

    print("\n✓ test_validate_signature_decorator PASSED")

if __name__ == "__main__":
    print("=" * 70)
    print("SIGNATURE VALIDATION TESTS")
    print("=" * 70 + "\n")

    try:
        test_defensive_function_with_extra_kwargs()
        test_defensive_function_with_optional_param()
        test_defensive_class_init_with_extra_kwargs()
        test_signature_validator_basic_functionality()
        test_validate_signature_decorator()

        print("\n" + "=" * 70)
        print("ALL TESTS PASSED ✓")
        print("=" * 70)
        print("\nSignature validation system is working correctly!")
        print("The defensive programming fixes prevent signature mismatch crashes.")

    except Exception as e:
        print(f"\n✗ TEST FAILED: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```

```
===== FILE: tests/test_sota_pa_coverage_integration.py =====
"""Integration tests for SOTA PA coverage pipeline.

Tests all 5 components:
1. Soft-alias pattern
2. Quality metrics monitoring
3. Intelligent fallback fusion
4. Hard calibration gates
5. Cache invalidation
"""

import pytest

from saaaaaa.core.orchestrator.signal_loader import build_all_signal_packs
from saaaaaa.core.orchestrator.signals import SignalPack
from saaaaaa.core.orchestrator.signal_aliasing import (
    canonicalize_signal_fingerprint,
    upgrade_legacy_fingerprints,
    validate_fingerprint_uniqueness,
    build_fingerprint_index,
)
from saaaaaa.core.orchestrator.signal_quality_metrics import (
    compute_signal_quality_metrics,
    analyze_coverage_gaps,
    generate_quality_report,
)
from saaaaaa.core.orchestrator.signal_fallback_fusion import (
    apply_intelligent_fallback_fusion,
    FusionStrategy,
    select_fusion_candidates,
)
from saaaaaa.core.orchestrator.signal_calibration_gate import (
    run_calibration_gates,
    CalibrationGateConfig,
    GateSeverity,
)
from saaaaaa.core.orchestrator.signal_cache_invalidation import (
    SignalPackCache,
    build_cache_key,
    validate_cache_integrity,
)


class TestSoftAliasPattern:
    """Test soft-alias pattern for fingerprint canonicalization."""

    def test_canonicalize_fingerprint_is_deterministic(self):
        """Test that canonicalize_signal_fingerprint produces deterministic results."""
        pack = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["tierras", "territorio"],
            indicators=["reforma agraria"],
            regex=["\\btierras?\\b"],
            entities=["ANT"],
            thresholds={"min_confidence": 0.77},
            metadata={"original_policy_area": "PA07"},
        )

        fp1 = canonicalize_signal_fingerprint(pack)
        fp2 = canonicalize_signal_fingerprint(pack)

        assert fp1 == fp2, "Fingerprint should be deterministic"
        assert len(fp1) == 32, "Fingerprint should be 32 chars"

    def test_canonicalize_fingerprint_changes_with_content(self):
        """Test that fingerprint changes when content changes."""
        pack1 = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["tierras"],
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
            metadata={"original_policy_area": "PA07"},
        )

        pack2 = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["tierras", "territorio"],  # Added pattern
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
            metadata={"original_policy_area": "PA07"},
        )

        fp1 = canonicalize_signal_fingerprint(pack1)
        fp2 = canonicalize_signal_fingerprint(pack2)

        assert fp1 != fp2, "Fingerprint should change when content changes"

    def test_upgrade_legacy_fingerprints(self):
        """Test upgrading legacy static fingerprints to canonical."""
        signal_packs = build_all_signal_packs()

        # Check that PA07-PA10 have legacy fingerprints
        legacy_fps = {
            signal_packs[pa].source_fingerprint
            for pa in ["PA07", "PA08", "PA09", "PA10"]
        }
        assert any("pa0" in fp for fp in legacy_fps), "Should have legacy fingerprints"

        # Upgrade
        upgraded_packs = upgrade_legacy_fingerprints(signal_packs)

        # Check that fingerprints are now canonical (content-based)
        for pa in ["PA07", "PA08", "PA09", "PA10"]:
            pack = upgraded_packs[pa]
            assert len(pack.source_fingerprint) == 32, "Should be 32-char hash"
            assert pack.metadata["migration"]["upgraded_to_canonical"]

    def test_validate_fingerprint_uniqueness(self):
        """Test fingerprint uniqueness validation."""
        signal_packs = build_all_signal_packs()
        upgraded_packs = upgrade_legacy_fingerprints(signal_packs)

        result = validate_fingerprint_uniqueness(upgraded_packs)

        assert result["is_valid"], "All fingerprints should be unique"
        assert result["total_fingerprints"] == 10, "Should have 10 fingerprints"
        assert len(result["duplicates"]) == 0, "Should have no duplicates"

    def test_build_fingerprint_index(self):
        """Test building fingerprint index."""
        signal_packs = build_all_signal_packs()
        upgraded_packs = upgrade_legacy_fingerprints(signal_packs)

        index = build_fingerprint_index(upgraded_packs)

        assert len(index) >= 10, "Should have at least 10 entries"

        # Check that we can resolve PA07 by fingerprint
        pa07_fp = upgraded_packs["PA07"].source_fingerprint
        assert index[pa07_fp] == "PA07"


class TestQualityMetricsMonitoring:
    """Test quality metrics monitoring for PA coverage."""

    def test_compute_signal_quality_metrics(self):
        """Test computing quality metrics for a signal pack."""
        signal_packs = build_all_signal_packs()

        metrics = compute_signal_quality_metrics(signal_packs["PA07"], "PA07")

        assert metrics.policy_area_id == "PA07"
        assert metrics.pattern_count > 0
        assert metrics.coverage_tier in ("EXCELLENT", "GOOD", "ADEQUATE", "SPARSE")
        assert 0.0 <= metrics.threshold_min_confidence <= 1.0
        assert 0.0 <= metrics.entity_coverage_ratio <= 1.0

    def test_analyze_coverage_gaps(self):
        """Test coverage gap analysis."""
        signal_packs = build_all_signal_packs()
        metrics_by_pa = {
            pa: compute_signal_quality_metrics(pack, pa)
            for pa, pack in signal_packs.items()
        }

        gap_analysis = analyze_coverage_gaps(metrics_by_pa)

        assert gap_analysis.gap_severity in (
            "CRITICAL", "SEVERE", "MODERATE", "MINOR", "NEGLIGIBLE"
        )
        assert len(gap_analysis.high_coverage_pas) > 0
        assert len(gap_analysis.low_coverage_pas) > 0
        assert gap_analysis.coverage_delta >= 0

    def test_generate_quality_report(self):
        """Test generating comprehensive quality report."""
        signal_packs = build_all_signal_packs()
        metrics_by_pa = {
            pa: compute_signal_quality_metrics(pack, pa)
            for pa, pack in signal_packs.items()
        }

        report = generate_quality_report(metrics_by_pa)

        assert "summary" in report
        assert "by_policy_area" in report
        assert "coverage_gap_analysis" in report
        assert "quality_gates" in report

        assert report["summary"]["total_policy_areas"] == 10
        assert report["summary"]["total_patterns"] > 0


class TestIntelligentFallbackFusion:
    """Test intelligent fallback fusion for PA07-PA10."""

    def test_select_fusion_candidates(self):
        """Test selecting fusion candidates based on similarity."""
        source_patterns = ["tierras", "territorio", "reforma agraria", "catastro"]
        target_patterns = ["tierras"]
        strategy = FusionStrategy(similarity_threshold=0.30, max_fusion_ratio=0.50)

        candidates = select_fusion_candidates(source_patterns, target_patterns, strategy)

        # Should not include "tierras" (already in target)
        assert "tierras" not in candidates
        # Should have at most max_fusion_ratio * len(target_patterns) candidates
        assert len(candidates) <= int(len(target_patterns) * strategy.max_fusion_ratio)

    def test_apply_intelligent_fallback_fusion(self):
        """Test applying intelligent fallback fusion."""
        signal_packs = build_all_signal_packs()
        metrics_by_pa = {
            pa: compute_signal_quality_metrics(pack, pa)
            for pa, pack in signal_packs.items()
        }

        # Apply fusion
        fused_packs = apply_intelligent_fallback_fusion(signal_packs, metrics_by_pa)

        # Check that low-coverage PAs were augmented
        for pa in ["PA07", "PA08", "PA09", "PA10"]:
            if metrics_by_pa[pa].coverage_tier in ("SPARSE", "ADEQUATE"):
                original_count = len(signal_packs[pa].patterns)
                fused_count = len(fused_packs[pa].patterns)

                # Fused pack should have more patterns (or same if no fusion needed)
                assert fused_count >= original_count

                # Check fusion metadata
                if fused_count > original_count:
                    assert "fusion" in fused_packs[pa].metadata
                    assert fused_packs[pa].metadata["fusion"]["fusion_enabled"]


class TestHardCalibrationGates:
    """Test hard calibration gates."""

    def test_run_calibration_gates_all_pass(self):
        """Test calibration gates with relaxed config."""
        signal_packs = build_all_signal_packs()
        upgraded_packs = upgrade_legacy_fingerprints(signal_packs)
        metrics_by_pa = {
            pa: compute_signal_quality_metrics(pack, pa)
            for pa, pack in upgraded_packs.items()
        }

        # Relaxed config for testing
        config = CalibrationGateConfig(
            min_patterns_per_pa=5,
            min_confidence_threshold=0.65,
            max_threshold_drift=0.20,
        )

        result = run_calibration_gates(upgraded_packs, metrics_by_pa, config)

        # Should pass with relaxed config
        assert isinstance(result.passed, bool)
        assert len(result.violations) >= 0

    def test_calibration_gates_detect_violations(self):
        """Test that calibration gates detect violations."""
        # Create a minimal pack that violates gates
        minimal_pack = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["test"],  # Only 1 pattern (violates min_patterns_per_pa)
            indicators=[],
            regex=[],
            entities=[],
            thresholds={"min_confidence": 0.50},  # Low threshold
            metadata={"original_policy_area": "PA99"},
        )
        signal_packs = {"PA99": minimal_pack}
        metrics = compute_signal_quality_metrics(minimal_pack, "PA99")
        metrics_by_pa = {"PA99": metrics}

        config = CalibrationGateConfig(
            min_patterns_per_pa=10,
            min_confidence_threshold=0.70,
        )

        result = run_calibration_gates(signal_packs, metrics_by_pa, config)

        # Should fail
        assert not result.passed
        assert result.has_errors

        # Check specific violations
        error_violations = result.get_violations_by_severity(GateSeverity.ERROR)
        assert len(error_violations) > 0

        # Should have pattern_coverage violation
        violation_names = [v.gate_name for v in error_violations]
        assert "pattern_coverage" in violation_names
        assert "confidence_threshold_too_low" in violation_names


class TestCacheInvalidation:
    """Test cache invalidation for data integrity."""

    def test_cache_put_and_get(self):
        """Test basic cache put/get operations."""
        cache = SignalPackCache(max_size=10)

        pack = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["test"],
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
            metadata={"original_policy_area": "PA01"},
        )

        key = build_cache_key("PA01", pack)

        # Put
        cache.put(key, "PA01", pack)

        # Get
        cached_pack = cache.get(key)
        assert cached_pack is not None
        assert cached_pack.patterns == ["test"]

    def test_cache_expiration(self):
        """Test cache TTL expiration."""
        cache = SignalPackCache(max_size=10)

        pack = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["test"],
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
            ttl_s=1,  # 1 second TTL
            metadata={"original_policy_area": "PA01"},
        )
        key = build_cache_key("PA01", pack)
        cache.put(key, "PA01", pack, ttl_seconds=0.1)  # 100ms TTL

        # Should be cached immediately
        cached = cache.get(key)
        assert cached is not None

        # Wait for expiration
        import time
        time.sleep(0.2)

        # Should be expired
        expired = cache.get(key)
        assert expired is None

    def test_cache_invalidation(self):
        """Test manual cache invalidation."""
        cache = SignalPackCache(max_size=10)

        pack = SignalPack(
            version="1.0.0",
            policy_area="fiscal",
            patterns=["test"],
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
            metadata={"original_policy_area": "PA01"},
        )

        key = build_cache_key("PA01", pack)
        cache.put(key, "PA01", pack)

        # Invalidate
        invalidated = cache.invalidate(key, "test_invalidation")
        assert invalidated

        # Should be gone
        cached = cache.get(key)
        assert cached is None

    def test_cache_warm_and_validate_integrity(self):
        """Test cache warming and integrity validation."""
        cache = SignalPackCache(max_size=100)

        signal_packs = build_all_signal_packs()
        upgraded_packs = upgrade_legacy_fingerprints(signal_packs)

        # Warm cache
        warmed_count = cache.warm_cache(upgraded_packs)
        assert warmed_count == len(upgraded_packs)

        # Validate integrity
        result = validate_cache_integrity(cache, upgraded_packs)
        assert result["is_valid"]
        assert len(result["mismatched_entries"]) == 0


class TestEndToEndIntegration:
    """End-to-end integration tests."""

    def test_complete_sota_pipeline(self):
        """Test complete SOTA pipeline with all 5 components."""

        # 1. Load signal packs
        signal_packs = build_all_signal_packs()
        assert len(signal_packs) == 10
        # 2. Upgrade legacy fingerprints (soft-alias)
        signal_packs = upgrade_legacy_fingerprints(signal_packs)
        fingerprint_validation = validate_fingerprint_uniqueness(signal_packs)
        assert fingerprint_validation["is_valid"]

        # 3. Compute quality metrics
        metrics_by_pa = {
            pa: compute_signal_quality_metrics(pack, pa)
            for pa, pack in signal_packs.items()
        }
        assert len(metrics_by_pa) == 10

        # 4. Analyze coverage gaps
        gap_analysis = analyze_coverage_gaps(metrics_by_pa)
        assert gap_analysis.gap_severity in (
            "CRITICAL", "SEVERE", "MODERATE", "MINOR", "NEGLIGIBLE"
        )

        # 5. Apply intelligent fallback fusion (if needed)
        if gap_analysis.requires_fallback_fusion:
            signal_packs = apply_intelligent_fallback_fusion(
                signal_packs,
                metrics_by_pa,
            )

            # Recompute metrics after fusion
            metrics_by_pa = {
                pa: compute_signal_quality_metrics(pack, pa)
                for pa, pack in signal_packs.items()
            }

        # 6. Run calibration gates
        config = CalibrationGateConfig(
            min_patterns_per_pa=8,  # Relaxed for PA07-PA10
            min_confidence_threshold=0.70,
        )
        gate_result = run_calibration_gates(signal_packs, metrics_by_pa, config)

        # Should pass after fusion (or have minimal violations)
        assert isinstance(gate_result.passed, bool)

        # 7. Warm cache
        cache = SignalPackCache(max_size=100)
        warmed_count = cache.warm_cache(signal_packs)
        assert warmed_count == 10

        # 8. Validate cache integrity
        cache_validation = validate_cache_integrity(cache, signal_packs)
        assert cache_validation["is_valid"]

        # 9. Generate final quality report
        quality_report = generate_quality_report(metrics_by_pa)
        assert quality_report["summary"]["total_policy_areas"] == 10
        assert quality_report["quality_gates"]["all_pas_have_patterns"]


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_spc_adapter.py =====
"""Tests for SPC Adapter.

Tests conversion from Smart Policy Chunks (SPC) / Canon Policy Package (CPP)
to PreprocessedDocument. These tests directly test SPCAdapter without wrappers.
"""

from __future__ import annotations
```

```
import pytest

from saaaaaa.utils.spc_adapter import SPCAdapter, SPCAdapterError,
adapt_spc_to_orchestrator
from saaaaaa.processing.cpp_ingestion.models import (
    CanonPolicyPackage,
    Chunk,
    ChunkGraph,
    ChunkResolution,
    Confidence,
    PolicyFacet,
    PolicyManifest,
    ProvenanceMap,
    QualityMetrics,
    TextSpan,
    TimeFacet,
    GeoFacet,
    IntegrityIndex,
    Budget,  # Fixed: was BudgetInfo, actual class name is Budget
    KPI,     # Fixed: was KPIInfo, actual class name is KPI
)

# Compatibility aliases for test code
BudgetInfo = Budget
KPIInfo = KPI


def create_test_chunk(
    chunk_id: str,
    text: str,
    start: int,
    end: int,
    resolution: ChunkResolution = ChunkResolution.MICRO,
    budget: BudgetInfo | None = None,
    kpi: KPIInfo | None = None,
) -> Chunk:
    """Create a test chunk."""
    return Chunk(
        id=chunk_id,
        bytes_hash=f"hash_{chunk_id}",
        text_span=TextSpan(start=start, end=end),
        resolution=resolution,
        text=text,
        policy_facets=PolicyFacet(
            axes=["Eje1", "Eje2"],
            programs=["Programa A"],
            projects=["Proyecto X"]
        ),
        time_facets=TimeFacet(years=[2024, 2025]),
        geo_facets=GeoFacet(territories=["Bogotá", "Antioquia"]),
        confidence=Confidence(layout=1.0, ocr=0.98, typing=0.95),
        budget=budget,
        kpi=kpi,
    )


def create_test_spc(chunks: list[Chunk]) -> CanonPolicyPackage:
    """Create a test SPC/CPP with given chunks."""
    chunk_graph = ChunkGraph()
    for chunk in chunks:
        chunk_graph.add_chunk(chunk)

    return CanonPolicyPackage(
        schema_version="SPC-2025.1",
        policy_manifest=PolicyManifest(
            axes=3,
            programs=10,
            projects=25,
            years=[2024, 2025],
            territories=["Bogotá", "Antioquia", "Valle"],
            indicators=15,
            budget_rows=50,
        ),
        chunk_graph=chunk_graph,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(
            provenance_completeness=1.0,
            structural_consistency=0.95,
            boundary_f1=0.88,
            kpi_linkage_rate=0.75,
            budget_consistency_score=0.92,
            temporal_robustness=0.85,
            chunk_context_coverage=0.90,
        ),
        integrity_index=IntegrityIndex(
            blake3_root="test_root_hash_spc",
            chunk_hashes={
                "chunk_1": "hash1",
                "chunk_2": "hash2",
                "chunk_3": "hash3",
            }
        ),
    )


def test_spc_adapter_initialization() -> None:
    """Test SPC adapter initialization."""
    adapter = SPCAdapter()

    metrics = adapter.get_metrics()
    assert metrics["conversions_count"] == 0


def test_to_preprocessed_document_basic() -> None:
    """Test basic SPC to PreprocessedDocument conversion."""
    chunks = [
        create_test_chunk("chunk_1", "First chunk text.", 0, 17),
        create_test_chunk("chunk_2", "Second chunk text.", 18, 36),
        create_test_chunk("chunk_3", "Third chunk text.", 37, 54),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_doc_spc")

    assert doc.document_id == "test_doc_spc"
    assert "First chunk text." in doc.raw_text
    assert "Second chunk text." in doc.raw_text
    assert "Third chunk text." in doc.raw_text
    assert len(doc.sentences) == 3
    assert len(doc.metadata["chunks"]) == 3


def test_to_preprocessed_document_with_schema_version() -> None:
    """Test that schema_version is preserved in metadata."""
    chunks = [
        create_test_chunk("chunk_1", "Test content.", 0, 13),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_schema")
    assert doc.metadata["schema_version"] == "SPC-2025.1"
    assert doc.metadata["adapter_source"] == "cpp_adapter.CPPAdapter"


def test_to_preprocessed_document_with_policy_manifest() -> None:
    """Test that policy manifest is included in metadata."""
    chunks = [
        create_test_chunk("chunk_1", "Policy content.", 0, 15),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_manifest")

    assert "policy_manifest" in doc.metadata
    manifest = doc.metadata["policy_manifest"]
    assert manifest["axes"] == 3
    assert manifest["programs"] == 10
    assert manifest["projects"] == 25
    assert manifest["years"] == [2024, 2025]
    assert manifest["territories"] == ["Bogotá", "Antioquia", "Valle"]
    assert manifest["indicators"] == 15
    assert manifest["budget_rows"] == 50


def test_to_preprocessed_document_with_quality_metrics() -> None:
    """Test that quality metrics are included in metadata."""
    chunks = [
        create_test_chunk("chunk_1", "Quality test.", 0, 13),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_quality")

    assert "quality_metrics" in doc.metadata
    metrics = doc.metadata["quality_metrics"]
    assert metrics["provenance_completeness"] == 1.0
    assert metrics["structural_consistency"] == 0.95
    assert metrics["boundary_f1"] == 0.88
    assert metrics["kpi_linkage_rate"] == 0.75
    assert metrics["budget_consistency_score"] == 0.92
    assert metrics["temporal_robustness"] == 0.85
    assert metrics["chunk_context_coverage"] == 0.90


def test_to_preprocessed_document_with_integrity_index() -> None:
    """Test that integrity index is included in metadata."""
    chunks = [
        create_test_chunk("chunk_1", "Integrity test.", 0, 15),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_integrity")

    assert "integrity_index" in doc.metadata
    index = doc.metadata["integrity_index"]
    assert index["blake3_root"] == "test_root_hash_spc"
    assert index["chunk_hashes_count"] == 3


def test_to_preprocessed_document_with_budget_data() -> None:
    """Test that budget data is extracted into tables."""
    budget1 = BudgetInfo(
        source="Fuente A",
        use="Destino A",
        amount=1000000.0,
        year=2024,
        currency="COP"
    )
    budget2 = BudgetInfo(
        source="Fuente B",
        use="Destino B",
        amount=2500000.0,
        year=2025,
        currency="COP"
    )

    chunks = [
        create_test_chunk("chunk_1", "Budget chunk 1.", 0, 15, budget=budget1),
        create_test_chunk("chunk_2", "Budget chunk 2.", 16, 31, budget=budget2),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_budget")

    assert len(doc.tables) == 2
    assert doc.tables[0]["source"] == "Fuente A"
    assert doc.tables[0]["amount"] == 1000000.0
    assert doc.tables[0]["year"] == 2024
    assert doc.tables[1]["source"] == "Fuente B"
    assert doc.tables[1]["amount"] == 2500000.0


def test_to_preprocessed_document_chunk_ordering() -> None:
    """Test that chunks are ordered by text_span.start."""
    chunks = [
        create_test_chunk("chunk_3", "Third.", 100, 106),
        create_test_chunk("chunk_1", "First.", 0, 6),
        create_test_chunk("chunk_2", "Second.", 50, 57),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_order")

    # Check that sentences are in correct order
    assert doc.sentences[0]["text"] == "First."
    assert doc.sentences[1]["text"] == "Second."
    assert doc.sentences[2]["text"] == "Third."

    # Check raw text is concatenated in order
    assert doc.raw_text == "First. Second. Third."


def test_to_preprocessed_document_resolution_filter() -> None:
    """Test filtering by chunk resolution."""
    chunks = [
        create_test_chunk("chunk_1", "Micro 1.", 0, 8, ChunkResolution.MICRO),
        create_test_chunk("chunk_2", "Meso 1.", 9, 16, ChunkResolution.MESO),
        create_test_chunk("chunk_3", "Micro 2.", 17, 25, ChunkResolution.MICRO),
        create_test_chunk("chunk_4", "Macro 1.", 26, 34, ChunkResolution.MACRO),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    # Filter for MICRO chunks only
    doc = adapter.to_preprocessed_document(
        spc,
        document_id="test_filter",
        preserve_chunk_resolution=ChunkResolution.MICRO
    )

    assert len(doc.sentences) == 2
    assert doc.sentences[0]["text"] == "Micro 1."
    assert doc.sentences[1]["text"] == "Micro 2."
    assert doc.sentences[0]["resolution"] == "micro"
    assert doc.sentences[1]["resolution"] == "micro"


def test_to_preprocessed_document_chunk_metadata() -> None:
    """Test that chunk metadata is preserved."""
    kpi = KPIInfo(
        name="Tasa de cobertura",
        baseline=75.0,
        target=90.0,
        unit="%"
    )

    chunks = [
        create_test_chunk("chunk_1", "Chunk with KPI.", 0, 15, kpi=kpi),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_metadata")

    chunk_meta = doc.metadata["chunks"][0]
    assert chunk_meta["id"] == "chunk_1"
    assert chunk_meta["resolution"] == "micro"
    assert chunk_meta["text_span"]["start"] == 0
    assert chunk_meta["text_span"]["end"] == 15
    assert chunk_meta["has_kpi"] is True
    assert chunk_meta["has_budget"] is False
    assert chunk_meta["confidence"]["layout"] == 1.0
    assert chunk_meta["confidence"]["ocr"] == 0.98
    assert chunk_meta["confidence"]["typing"] == 0.95


def test_to_preprocessed_document_empty_chunk_graph_error() -> None:
    """Test error when chunk graph is empty."""
    spc = CanonPolicyPackage(
        schema_version="SPC-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=ChunkGraph(),  # Empty
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="test"),
    )

    adapter = SPCAdapter()

    with pytest.raises(SPCAdapterError) as exc_info:
        adapter.to_preprocessed_document(spc, document_id="test")

    assert "chunk graph is empty" in str(exc_info.value).lower()


def test_to_preprocessed_document_none_spc_error() -> None:
    """Test error when SPC is None."""
    adapter = SPCAdapter()

    with pytest.raises(SPCAdapterError) as exc_info:
        adapter.to_preprocessed_document(None, document_id="test")
    assert "cannot convert none" in str(exc_info.value).lower()


def test_to_preprocessed_document_no_chunks_with_resolution_error() -> None:
    """Test error when no chunks match the requested resolution."""
    chunks = [
        create_test_chunk("chunk_1", "Micro only.", 0, 11, ChunkResolution.MICRO),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    with pytest.raises(SPCAdapterError) as exc_info:
        adapter.to_preprocessed_document(
            spc,
            document_id="test",
            preserve_chunk_resolution=ChunkResolution.MACRO
        )

    error_msg = str(exc_info.value).lower()
    assert "no chunks found" in error_msg
    assert "macro" in error_msg


def test_adapt_spc_to_orchestrator_convenience_function() -> None:
    """Test the convenience function."""
    chunks = [
        create_test_chunk("chunk_1", "Convenience test.", 0, 17),
    ]

    spc = create_test_spc(chunks)

    doc = adapt_spc_to_orchestrator(spc, document_id="test_convenience")

    assert doc.document_id == "test_convenience"
    assert "Convenience test." in doc.raw_text


def test_adapter_metrics_tracking() -> None:
    """Test that adapter tracks conversion metrics."""
    adapter = SPCAdapter()

    chunks = [create_test_chunk("chunk_1", "Test.", 0, 5)]
    spc = create_test_spc(chunks)

    # Perform multiple conversions
    adapter.to_preprocessed_document(spc, document_id="doc1")
    adapter.to_preprocessed_document(spc, document_id="doc2")
    adapter.to_preprocessed_document(spc, document_id="doc3")

    metrics = adapter.get_metrics()
    assert metrics["conversions_count"] == 3


def test_provenance_completeness_calculation() -> None:
    """Test provenance completeness calculation."""
    # Create chunks with provenance
    chunk1 = create_test_chunk("chunk_1", "With provenance.", 0, 16)
    chunk1.provenance = {"page": 1, "tokens": [0, 1, 2]}

    chunk2 = create_test_chunk("chunk_2", "No provenance.", 17, 31)
    chunk2.provenance = None

    chunk3 = create_test_chunk("chunk_3", "With provenance.", 32, 48)
    chunk3.provenance = {"page": 2, "tokens": [3, 4, 5]}

    chunks = [chunk1, chunk2, chunk3]
    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    doc = adapter.to_preprocessed_document(spc, document_id="test_provenance")

    # 2 out of 3 chunks have provenance = 2/3 ≈ 0.667
    assert abs(doc.metadata["provenance_completeness"] - 0.667) < 0.01

    # Check individual chunk provenance flags
    chunk_meta = doc.metadata["chunks"]
    assert chunk_meta[0]["has_provenance"] is True
    assert chunk_meta[1]["has_provenance"] is False
    assert chunk_meta[2]["has_provenance"] is True


def test_document_id_auto_generation() -> None:
    """Test automatic document ID generation from chunk IDs."""
    chunks = [
        create_test_chunk("mydoc_chunk_1", "Content.", 0, 8),
    ]

    spc = create_test_spc(chunks)
    adapter = SPCAdapter()

    # Don't provide document_id
    doc = adapter.to_preprocessed_document(spc)

    # Should extract 'mydoc' from first chunk ID
    assert doc.document_id == "mydoc"

===== FILE: tests/test_spc_adapter_integration.py =====
"""Integration test for SPC Adapter.

Tests direct SPCAdapter usage (not through CPPAdapter wrapper).
Verifies full compatibility between SPC ingestion and orchestrator.
"""

from __future__ import annotations

import pytest

from saaaaaa.utils.spc_adapter import SPCAdapter, SPCAdapterError
from saaaaaa.processing.cpp_ingestion.models import (
    CanonPolicyPackage,
    Chunk,
    ChunkGraph,
    ChunkResolution,
    Confidence,
    PolicyFacet,
    TextSpan,
)


def test_spc_adapter_basic_conversion():
    """Test basic SPCAdapter conversion from CanonPolicyPackage to
PreprocessedDocument."""
    # Create test chunks
    chunk1 = Chunk(
        id="chunk_001",
        text="This is the first policy chunk.",
        text_span=TextSpan(start=0, end=32),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_001",
        confidence=Confidence(layout=0.95, ocr=0.98, typing=0.99),
        policy_facets=PolicyFacet(programs=["Program A"], axes=["Axis 1"]),
    )

    chunk2 = Chunk(
        id="chunk_002",
        text="This is the second policy chunk.",
        text_span=TextSpan(start=33, end=66),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_002",
        confidence=Confidence(layout=0.97, ocr=0.96, typing=0.98),
        policy_facets=PolicyFacet(programs=["Program B"], axes=["Axis 2"]),
    )

    # Create ChunkGraph
    chunk_graph = ChunkGraph(chunks={"chunk_001": chunk1, "chunk_002": chunk2})

    # Create CanonPolicyPackage
    cpp = CanonPolicyPackage(
        schema_version="3.0.0",
        chunk_graph=chunk_graph,
    )

    # Convert using SPCAdapter
    adapter = SPCAdapter()
    doc = adapter.to_preprocessed_document(cpp, document_id="test_doc")

    # Verify PreprocessedDocument structure
    assert doc.document_id == "test_doc"
    assert doc.raw_text == "This is the first policy chunk. This is the second policy
chunk."
    assert len(doc.sentences) == 2
    assert doc.sentences[0]["text"] == "This is the first policy chunk."
    assert doc.sentences[0]["chunk_id"] == "chunk_001"
    assert doc.sentences[0]["resolution"] == "MICRO"
    assert doc.sentences[1]["text"] == "This is the second policy chunk."

    # Verify metadata
    assert doc.metadata["adapter_source"] == "spc_adapter.SPCAdapter"
    assert doc.metadata["schema_version"] == "3.0.0"
    assert doc.metadata["chunk_count"] == 2
    assert doc.metadata["provenance_completeness"] == 0.0  # No provenance in test data

    # Verify chunk details in metadata
    assert len(doc.metadata["chunks"]) == 2
    assert doc.metadata["chunks"][0]["id"] == "chunk_001"
    assert doc.metadata["chunks"][0]["resolution"] == "MICRO"
    assert doc.metadata["chunks"][0]["confidence"]["layout"] == 0.95


def test_spc_adapter_empty_chunk_graph():
    """Test SPCAdapter error handling for empty chunk graph."""
    cpp = CanonPolicyPackage(
        schema_version="3.0.0",
        chunk_graph=ChunkGraph(chunks={}),
    )

    adapter = SPCAdapter()

    with pytest.raises(SPCAdapterError) as exc_info:
        adapter.to_preprocessed_document(cpp)

    assert "chunk graph is empty" in str(exc_info.value).lower()


def test_spc_adapter_chunk_ordering():
    """Test that chunks are ordered by text_span.start for determinism."""
    # Create chunks in non-sequential order
    chunk3 = Chunk(
        id="chunk_003",
        text="Third chunk.",
        text_span=TextSpan(start=200, end=212),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_003",
    )

    chunk1 = Chunk(
        id="chunk_001",
        text="First chunk.",
        text_span=TextSpan(start=0, end=12),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_001",
    )

    chunk2 = Chunk(
        id="chunk_002",
        text="Second chunk.",
        text_span=TextSpan(start=100, end=113),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_002",
    )

    # Add chunks in non-sequential order
    chunk_graph = ChunkGraph(chunks={
        "chunk_003": chunk3,
        "chunk_001": chunk1,
        "chunk_002": chunk2,
    })

    cpp = CanonPolicyPackage(
        schema_version="3.0.0",
        chunk_graph=chunk_graph,
    )

    adapter = SPCAdapter()
    doc = adapter.to_preprocessed_document(cpp)

    # Verify chunks are ordered by text_span.start
    assert doc.sentences[0]["chunk_id"] == "chunk_001"  # start=0
    assert doc.sentences[1]["chunk_id"] == "chunk_002"  # start=100
    assert doc.sentences[2]["chunk_id"] == "chunk_003"  # start=200
    assert doc.raw_text == "First chunk. Second chunk. Third chunk."


def test_spc_adapter_metrics():
    """Test SPCAdapter metrics tracking."""
    adapter = SPCAdapter()

    initial_metrics = adapter.get_metrics()
    assert initial_metrics["conversions_count"] == 0

    # Perform a conversion
    chunk = Chunk(
        id="chunk_001",
        text="Test chunk.",
        text_span=TextSpan(start=0, end=11),
        resolution=ChunkResolution.MICRO,
        bytes_hash="hash_001",
    )

    cpp = CanonPolicyPackage(
        schema_version="3.0.0",
        chunk_graph=ChunkGraph(chunks={"chunk_001": chunk}),
    )

    adapter.to_preprocessed_document(cpp)

    metrics = adapter.get_metrics()
    assert metrics["conversions_count"] == 1


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_spc_integration_complete.py =====
"""
Integration Test: SPC Phase 1 to Orchestrator Pipeline
=========================================================

This test verifies the complete integration between:
1. SmartPolicyChunk output from StrategicChunkingSystem
2. SmartChunkConverter conversion layer
3. CanonPolicyPackage format for orchestrator
4. PreprocessedDocument.ensure() acceptance

Tests the critical data flow identified in the audit.
"""

import pytest
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional, Tuple
from enum import Enum

from saaaaaa.processing.cpp_ingestion.models import (
    CanonPolicyPackage,
    ChunkGraph,
    Chunk,
    ChunkResolution,
    TextSpan,
    PolicyFacet,
    TimeFacet,
    GeoFacet,
    Confidence,
)
from saaaaaa.processing.spc_ingestion.converter import SmartChunkConverter
from saaaaaa.core.orchestrator.core import PreprocessedDocument


# Mock SmartPolicyChunk for testing (simplified version)
class ChunkType(Enum):
    DIAGNOSTICO = "diagnostico"
    ESTRATEGIA = "estrategia"
    METRICA = "metrica"
    FINANCIERO = "financiero"
    MIXTO = "mixto"


@dataclass
class MockCausalEvidence:
    dimension: str
    confidence: float


@dataclass
class MockPolicyEntity:
    text: str
    entity_type: str
    confidence: float


@dataclass
class MockStrategicContext:
    policy_intent: str
    implementation_phase: str
    geographic_scope: str
    temporal_horizon: str
    budget_linkage: str


@dataclass
```

```
class MockSmartPolicyChunk:
    """Simplified mock of SmartPolicyChunk for testing."""
    chunk_id: str
    document_id: str
    content_hash: str
    text: str
    normalized_text: str
    semantic_density: float
    section_hierarchy: List[str]
    document_position: Tuple[int, int]
    chunk_type: ChunkType
    causal_chain: List[MockCausalEvidence] = field(default_factory=list)
    policy_entities: List[MockPolicyEntity] = field(default_factory=list)
    related_chunks: List[Tuple[str, float]] = field(default_factory=list)
    confidence_metrics: Dict[str, float] = field(default_factory=dict)
    coherence_score: float = 0.8
    completeness_index: float = 0.85
    strategic_importance: float = 0.75
    information_density: float = 0.7
    actionability_score: float = 0.8
    strategic_context: Optional[MockStrategicContext] = None
    temporal_dynamics: Any = None


def create_mock_smart_chunks() -> List[MockSmartPolicyChunk]:
    """Create mock SmartPolicyChunk instances for testing."""
    chunks = []

    # Macro-level strategic chunk
    chunks.append(MockSmartPolicyChunk(
        chunk_id="strategic_axis_1",
        document_id="test_plan_2024",
        content_hash="hash_001",
        text="EJE ESTRATÉGICO 1: DESARROLLO SOCIAL. Visión 2028: Garantizar acceso
universal a educación y salud de calidad.",
        normalized_text="eje estrategico desarrollo social",
        semantic_density=0.85,
        section_hierarchy=["Desarrollo Social", "Objetivos Estratégicos"],
        document_position=(0, 120),
        chunk_type=ChunkType.ESTRATEGIA,
        causal_chain=[
            MockCausalEvidence(dimension="impactos", confidence=0.9)
        ],
        policy_entities=[
            MockPolicyEntity(text="Ministerio de Educación", entity_type="institution",
confidence=0.95)
        ],
        related_chunks=[("programa_educacion", 0.82)],
        confidence_metrics={"extraction_confidence": 0.95},
        coherence_score=0.92,
        completeness_index=0.88,
        strategic_importance=0.95,
        strategic_context=MockStrategicContext(
            policy_intent="Desarrollo Social Universal",
            implementation_phase="2024-2028",
            geographic_scope="Departamental",
            temporal_horizon="2024-2028",
            budget_linkage="$150,000 millones COP"
        )
    ))

    # Meso-level program chunk
    chunks.append(MockSmartPolicyChunk(
        chunk_id="programa_educacion",
        document_id="test_plan_2024",
        content_hash="hash_002",
        text="Programa: Educación para Todos. Objetivo: Cobertura del 95% en educación
primaria para 2026.",
        normalized_text="programa educacion para todos",
        semantic_density=0.78,
        section_hierarchy=["Desarrollo Social", "Educación", "Programa"],
        document_position=(121, 220),
        chunk_type=ChunkType.METRICA,
        causal_chain=[
            MockCausalEvidence(dimension="productos", confidence=0.85)
        ],
        policy_entities=[
            MockPolicyEntity(text="Secretaría de Educación", entity_type="institution",
confidence=0.90)
        ],
        related_chunks=[("strategic_axis_1", 0.82), ("proyecto_escuelas", 0.75)],
        confidence_metrics={"extraction_confidence": 0.90},
        coherence_score=0.85,
        completeness_index=0.82,
        strategic_importance=0.80
    ))

    # Micro-level project chunk
    chunks.append(MockSmartPolicyChunk(
        chunk_id="proyecto_escuelas",
        document_id="test_plan_2024",
        content_hash="hash_003",
        text="Proyecto: Construcción de 20 escuelas rurales. Presupuesto: $45,000
millones. Plazo: 2024-2026.",
        normalized_text="proyecto construccion escuelas rurales",
        semantic_density=0.82,
        section_hierarchy=["Desarrollo Social", "Educación", "Proyectos", "Escuelas
Rurales"],
        document_position=(221, 340),
        chunk_type=ChunkType.FINANCIERO,
        causal_chain=[
            MockCausalEvidence(dimension="actividades", confidence=0.88)
        ],
        policy_entities=[
            MockPolicyEntity(text="población rural", entity_type="beneficiary",
confidence=0.85)
        ],
        related_chunks=[("programa_educacion", 0.75)],
        confidence_metrics={"extraction_confidence": 0.92},
        coherence_score=0.87,
        completeness_index=0.90,
        strategic_importance=0.70,
        strategic_context=MockStrategicContext(
            policy_intent="Infraestructura Educativa Rural",
            implementation_phase="2024-2026",
            geographic_scope="Zonas Rurales",
            temporal_horizon="2024-2026",
            budget_linkage="$45,000 millones COP"
        )
    ))

    return chunks


class TestSPCIntegrationComplete:
    """Test complete SPC to Orchestrator integration."""

    def test_converter_initialization(self):
        """Test that SmartChunkConverter can be initialized."""
        converter = SmartChunkConverter()
        assert converter is not None
        assert hasattr(converter, 'convert_to_canon_package')

    def test_smart_chunk_to_canon_package_conversion(self):
        """Test conversion from SmartPolicyChunk to CanonPolicyPackage."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {
            'document_id': 'test_plan_2024',
            'title': 'Plan de Desarrollo 2024-2028',
            'version': 'v3.0'
        }
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Assert - Package structure
        assert isinstance(canon_package, CanonPolicyPackage)
        assert canon_package.schema_version == "SPC-2025.1"
        assert canon_package.chunk_graph is not None
        assert canon_package.policy_manifest is not None
        assert canon_package.quality_metrics is not None
        assert canon_package.integrity_index is not None

    def test_chunk_graph_structure(self):
        """Test that ChunkGraph is properly constructed."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)
        chunk_graph = canon_package.chunk_graph

        # Assert - Chunks
        assert isinstance(chunk_graph, ChunkGraph)
        assert len(chunk_graph.chunks) == 3
        assert "strategic_axis_1" in chunk_graph.chunks
        assert "programa_educacion" in chunk_graph.chunks
        assert "proyecto_escuelas" in chunk_graph.chunks

        # Assert - Chunk structure
        for chunk_id, chunk in chunk_graph.chunks.items():
            assert isinstance(chunk, Chunk)
            assert chunk.id == chunk_id
            assert chunk.text is not None and len(chunk.text) > 0
            assert isinstance(chunk.text_span, TextSpan)
            assert isinstance(chunk.resolution, ChunkResolution)
            assert chunk.bytes_hash is not None

        # Assert - Edges
        assert len(chunk_graph.edges) > 0
        for edge in chunk_graph.edges:
            assert len(edge) == 3  # (from, to, relation_type)
            assert edge[0] in chunk_graph.chunks  # from_id exists
            assert edge[1] in chunk_graph.chunks  # to_id exists

    def test_resolution_mapping(self):
        """Test that chunk_type is correctly mapped to ChunkResolution."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Assert - Resolution mapping
        strategic_chunk = canon_package.chunk_graph.chunks["strategic_axis_1"]
        assert strategic_chunk.resolution == ChunkResolution.MACRO  # ESTRATEGIA → MACRO

        program_chunk = canon_package.chunk_graph.chunks["programa_educacion"]
        assert program_chunk.resolution == ChunkResolution.MICRO  # METRICA → MICRO

        project_chunk = canon_package.chunk_graph.chunks["proyecto_escuelas"]
        assert project_chunk.resolution == ChunkResolution.MICRO  # FINANCIERO → MICRO

    def test_policy_facets_extraction(self):
        """Test that policy facets are extracted from SPC data."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Assert
        strategic_chunk = canon_package.chunk_graph.chunks["strategic_axis_1"]
        assert isinstance(strategic_chunk.policy_facets, PolicyFacet)
        assert len(strategic_chunk.policy_facets.axes) > 0

        project_chunk = canon_package.chunk_graph.chunks["proyecto_escuelas"]
        assert len(project_chunk.policy_facets.programs) > 0 or
len(project_chunk.policy_facets.projects) > 0

    def test_quality_metrics_calculation(self):
        """Test that quality metrics are calculated from SPC data."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)
        metrics = canon_package.quality_metrics

        # Assert
        assert metrics.provenance_completeness >= 0.0
        assert metrics.structural_consistency >= 0.0
        assert metrics.structural_consistency <= 1.0
        assert metrics.chunk_context_coverage >= 0.0

    def test_spc_rich_data_preservation(self):
        """Test that SPC rich data is preserved in metadata."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Assert - Rich data in metadata
        assert 'spc_rich_data' in canon_package.metadata
        assert 'spc_version' in canon_package.metadata
        assert canon_package.metadata['spc_version'] == 'SMART-CHUNK-3.0-FINAL'

        spc_data = canon_package.metadata['spc_rich_data']
        assert len(spc_data) == 3  # One entry per chunk

        # Check that quality scores are preserved
        for chunk_id in ['strategic_axis_1', 'programa_educacion', 'proyecto_escuelas']:
            assert chunk_id in spc_data
            chunk_spc_data = spc_data[chunk_id]
            assert 'coherence_score' in chunk_spc_data
            assert 'strategic_importance' in chunk_spc_data
            assert 'completeness_index' in chunk_spc_data

    def test_orchestrator_compatibility(self):
        """Test that CanonPolicyPackage is accepted by PreprocessedDocument.ensure()."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Act - Try to convert to PreprocessedDocument
        try:
            preprocessed = PreprocessedDocument.ensure(
                canon_package,
                document_id="test_plan_2024",
                use_spc_ingestion=True
            )

            # Assert - Orchestrator acceptance
            assert isinstance(preprocessed, PreprocessedDocument)
            assert preprocessed.document_id == "test_plan_2024"
            assert len(preprocessed.raw_text) > 0
            assert len(preprocessed.sentences) > 0

        except Exception as e:
            pytest.fail(f"PreprocessedDocument.ensure() rejected CanonPolicyPackage: {e}")

    def test_policy_manifest_completeness(self):
        """Test that PolicyManifest is properly populated."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)
        manifest = canon_package.policy_manifest

        # Assert
        assert len(manifest.axes) > 0
        assert len(manifest.programs) > 0
        assert len(manifest.projects) > 0
        # Years might be empty if not extracted from temporal dynamics

    def test_integrity_index_generation(self):
        """Test that IntegrityIndex is generated with valid hashes."""
        # Arrange
        smart_chunks = create_mock_smart_chunks()
        metadata = {'document_id': 'test_plan_2024', 'title': 'Test', 'version': 'v3.0'}
        converter = SmartChunkConverter()

        # Act
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)
        integrity = canon_package.integrity_index

        # Assert
        assert integrity.blake2b_root is not None
        assert len(integrity.blake2b_root) > 0
        assert len(integrity.chunk_hashes) == 3
        for chunk_id in ['strategic_axis_1', 'programa_educacion', 'proyecto_escuelas']:
            assert chunk_id in integrity.chunk_hashes

    def test_end_to_end_data_flow(self):
        """
        End-to-end test: SmartPolicyChunk → CanonPolicyPackage → PreprocessedDocument

        This test validates the complete data flow identified in the audit.
        """
        # Phase 1: SPC produces SmartPolicyChunks
        smart_chunks = create_mock_smart_chunks()
        metadata = {
            'document_id': 'test_plan_2024',
            'title': 'Plan de Desarrollo Departamental 2024-2028',
            'version': 'v3.0'
        }

        # Phase 2: Converter bridges to CanonPolicyPackage
        converter = SmartChunkConverter()
        canon_package = converter.convert_to_canon_package(smart_chunks, metadata)

        # Phase 3: Orchestrator accepts CanonPolicyPackage
        preprocessed = PreprocessedDocument.ensure(
            canon_package,
            document_id="test_plan_2024",
            use_spc_ingestion=True
        )

        # Validate end-to-end flow
        assert preprocessed.document_id == "test_plan_2024"
        assert len(preprocessed.sentences) == 3  # One per chunk
        assert 'spc_rich_data' in preprocessed.metadata
        assert preprocessed.metadata['chunk_count'] == 3

        # Verify data preservation through pipeline
        assert 'policy_manifest' in preprocessed.metadata
        assert 'quality_metrics' in preprocessed.metadata

        # Success: Complete pipeline verified!


if __name__ == '__main__':
    pytest.main([__file__, '-v'])

===== FILE: tests/test_spc_validation.py =====
"""Tests for SPC ingestion validation in orchestrator.

Tests the critical validation logic added to prevent empty document creation
and enforce SPC-only ingestion.
"""

import pytest
from saaaaaa.core.orchestrator.core import (
    Orchestrator,
    PreprocessedDocument,
    PhaseInstrumentation,
)
from saaaaaa.core.orchestrator.factory import build_processor
from saaaaaa.processing.cpp_ingestion.models import (
    CanonPolicyPackage,
    Chunk,
    ChunkGraph,
    ChunkResolution,
    Confidence,
    PolicyManifest,
    ProvenanceMap,
    QualityMetrics,
    TextSpan,
    IntegrityIndex,
)


def test_preprocessed_document_rejects_empty_raw_text():
    """Verify PreprocessedDocument.__post_init__ raises ValueError for empty raw_text."""
    with pytest.raises(ValueError, match="PreprocessedDocument cannot have empty
raw_text"):
        PreprocessedDocument(
            document_id="test",
            raw_text="",
            sentences=[],
            tables=[],
            metadata={}
        )


def test_preprocessed_document_rejects_whitespace_only():
    """Verify PreprocessedDocument.__post_init__ raises ValueError for whitespace-only
raw_text."""
    with pytest.raises(ValueError, match="PreprocessedDocument cannot have empty
raw_text"):
        PreprocessedDocument(
            document_id="test",
            raw_text="   \n  \t  ",
            sentences=[],
            tables=[],
            metadata={}
        )


def test_preprocessed_document_accepts_valid_text():
    """Verify PreprocessedDocument allows non-empty raw_text."""
    doc = PreprocessedDocument(
        document_id="test",
        raw_text="Valid content",
        sentences=[],
        tables=[],
        metadata={}
    )
    assert doc.raw_text == "Valid content"
    assert doc.document_id == "test"


def test_ensure_raises_with_legacy_ingestion():
    """Verify ValueError when use_spc_ingestion=False."""
    chunk = Chunk(
        id="test",
        bytes_hash="hash",
        text_span=TextSpan(0, 10),
        resolution=ChunkResolution.MICRO,
        text="Test text.",
        confidence=Confidence(layout=1.0),
    )
    graph = ChunkGraph()
    graph.add_chunk(chunk)
    cpp = CanonPolicyPackage(
        schema_version="CPP-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=graph,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="hash"),
    )

    with pytest.raises(ValueError, match="SPC ingestion is now required"):
        PreprocessedDocument.ensure(cpp, use_spc_ingestion=False)


def test_ensure_validates_empty_chunk_graph():
    """Verify ValueError when chunk_graph is empty."""
    empty_graph = ChunkGraph()  # No chunks
    cpp = CanonPolicyPackage(
        schema_version="CPP-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=empty_graph,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="hash"),
    )
    with pytest.raises(ValueError, match="chunk_graph is empty"):
        PreprocessedDocument.ensure(cpp, use_spc_ingestion=True)


def test_ensure_validates_chunk_graph_none():
    """Verify ValueError when chunk_graph is None."""
    cpp = CanonPolicyPackage(
        schema_version="CPP-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=None,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="hash"),
    )

    with pytest.raises(ValueError, match="chunk_graph attribute but it is None"):
        PreprocessedDocument.ensure(cpp, use_spc_ingestion=True)


def test_ensure_validates_empty_text_from_spc():
    """Verify ValueError for empty raw_text after SPC adaptation."""
    # Create chunk with empty text
    empty_chunk = Chunk(
        id="empty",
        bytes_hash="hash",
        text_span=TextSpan(0, 0),
        resolution=ChunkResolution.MICRO,
        text="",  # Empty!
        confidence=Confidence(layout=1.0),
    )
    graph = ChunkGraph()
    graph.add_chunk(empty_chunk)
    cpp = CanonPolicyPackage(
        schema_version="CPP-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=graph,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="hash"),
    )

    # Note: This will raise ValueError from __post_init__ since adapter creates
PreprocessedDocument
    with pytest.raises(ValueError, match="empty raw_text"):
        PreprocessedDocument.ensure(cpp, use_spc_ingestion=True)


def test_ensure_succeeds_with_valid_spc_document():
    """Verify successful conversion with valid SPC document."""
    chunk = Chunk(
        id="test",
        bytes_hash="hash",
        text_span=TextSpan(0, 50),
        resolution=ChunkResolution.MICRO,
        text="Valid policy document content for testing purposes.",
        confidence=Confidence(layout=1.0),
    )
    graph = ChunkGraph()
    graph.add_chunk(chunk)
    cpp = CanonPolicyPackage(
        schema_version="CPP-2025.1",
        policy_manifest=PolicyManifest(),
        chunk_graph=graph,
        provenance_map=ProvenanceMap(),
        quality_metrics=QualityMetrics(),
        integrity_index=IntegrityIndex(blake3_root="hash"),
    )
    doc = PreprocessedDocument.ensure(cpp, use_spc_ingestion=True)
    assert doc.raw_text
    assert len(doc.raw_text) > 0
    assert doc.metadata.get("chunk_count", 0) > 0


def test_ensure_rejects_unsupported_document_type():
    """Verify TypeError for documents without chunk_graph."""
    class FakeDocument:
        pass

    with pytest.raises(TypeError, match="Unsupported preprocessed document payload"):
        PreprocessedDocument.ensure(FakeDocument(), use_spc_ingestion=True)

===== FILE: tests/test_statistics_guards.py =====
"""Test statistics guards for edge cases in core.py."""
import statistics
import pytest


def test_variance_with_empty_list():
    """Test variance calculation with empty list."""
    normalized_values = []

    # Guard implementation
    variance = (
        statistics.variance(normalized_values)
        if len(normalized_values) >= 2
        else 0.0
    )

    assert variance == 0.0


def test_variance_with_single_value():
    """Test variance calculation with single value."""
    normalized_values = [0.5]

    # Guard implementation
    variance = (
        statistics.variance(normalized_values)
        if len(normalized_values) >= 2
        else 0.0
    )

    assert variance == 0.0


def test_variance_with_two_values():
    """Test variance calculation with two values."""
    normalized_values = [0.4, 0.6]

    # Guard implementation
    variance = (
        statistics.variance(normalized_values)
        if len(normalized_values) >= 2
        else 0.0
    )

    # Should calculate variance normally
    assert variance > 0.0
    assert variance == statistics.variance(normalized_values)


def test_variance_with_multiple_values():
    """Test variance calculation with multiple values."""
    normalized_values = [0.3, 0.5, 0.7, 0.9]
    # Guard implementation
    variance = (
        statistics.variance(normalized_values)
        if len(normalized_values) >= 2
        else 0.0
    )

    # Should calculate variance normally
    assert variance > 0.0
    assert variance == statistics.variance(normalized_values)


def test_variance_without_guard_raises():
    """Test that variance without guard raises error on single value."""
    normalized_values = [0.5]

    with pytest.raises(statistics.StatisticsError):
        statistics.variance(normalized_values)


def test_mean_with_empty_list():
    """Test mean calculation with empty list using guard."""
    values = []

    # Guard implementation
    mean = statistics.mean(values) if len(values) >= 1 else 0.0

    assert mean == 0.0


def test_mean_with_values():
    """Test mean calculation with values."""
    values = [1.0, 2.0, 3.0]

    # Guard implementation
    mean = statistics.mean(values) if len(values) >= 1 else 0.0

    assert mean == 2.0

===== FILE: tests/test_strategic_wiring.py =====
#!/usr/bin/env python3
"""
Strategic High-Level Wiring Validation Test
============================================

This test validates the high-level wiring and integration across
all strategic self-contained files mentioned in the requirements:

- demo_macro_prompts.py
- verify_complete_implementation.py
- validation_engine.py
- validate_system.py
- seed_factory.py
- qmcm_hooks.py
- meso_cluster_analysis.py
- macro_prompts.py
- json_contract_loader.py
- evidence_registry.py
- document_ingestion.py
- scoring.py
- recommendation_engine.py
- orchestrator.py
- micro_prompts.py
- coverage_gate.py
- scripts/bootstrap_validate.py
- validation/predicates.py
- validation/golden_rule.py
- validation/architecture_validator.py

Purpose: AUDIT, ENSURE, FORCE, GUARANTEE, and SUSTAIN high-level wiring
"""

import unittest
import pytest
from pathlib import Path

# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="outdated - use test_system_audit.py")

# Add parent directory and src to path
root_dir = Path(__file__).parent.parent

class TestStrategicWiring(unittest.TestCase):
    """Test suite for strategic file wiring validation."""

    def test_all_strategic_files_exist(self):
        """Verify all strategic files exist and are accessible."""
        strategic_files = [
            "demo_macro_prompts.py",
            "verify_complete_implementation.py",
            "validation_engine.py",
            "validate_system.py",
            "seed_factory.py",
            "qmcm_hooks.py",
            "meso_cluster_analysis.py",
            "macro_prompts.py",
            "json_contract_loader.py",
            "evidence_registry.py",
            "document_ingestion.py",
            "scoring.py",
            "recommendation_engine.py",
            "orchestrator.py",
            "micro_prompts.py",
            "coverage_gate.py",
            "scripts/bootstrap_validate.py",
            "validation/predicates.py",
            "validation/golden_rule.py",
            "validation/architecture_validator.py"
        ]

        root = Path(__file__).parent.parent
        for file_path in strategic_files:
            full_path = root / file_path
            self.assertTrue(
                full_path.exists(),
                f"Strategic file not found: {file_path}"
            )

    def test_provenance_includes_all_strategic_files(self):
        """Verify provenance.csv includes all strategic files."""
        root = Path(__file__).parent.parent
        # Check both root and data/ directory for provenance.csv
        provenance_path = root / "provenance.csv"
        if not provenance_path.exists():
            provenance_path = root / "data" / "provenance.csv"

        self.assertTrue(provenance_path.exists(), "provenance.csv not found")

        with open(provenance_path) as f:
            provenance_content = f.read()

        strategic_files_to_track = [
            "demo_macro_prompts.py",
            "verify_complete_implementation.py",
            "validation_engine.py",
            "validate_system.py",
            "seed_factory.py",
            "qmcm_hooks.py",
            "meso_cluster_analysis.py",
            "macro_prompts.py",
            "json_contract_loader.py",
            "evidence_registry.py",
            "document_ingestion.py",
            "scoring.py",
            "recommendation_engine.py",
            "orchestrator.py",
            "micro_prompts.py",
            "coverage_gate.py"
        ]

        for file_name in strategic_files_to_track:
            self.assertIn(
                file_name,
                provenance_content,
                f"Strategic file {file_name} not tracked in provenance.csv"
            )

    def test_validation_engine_imports(self):
        """Verify validation_engine.py imports correctly."""
        try:
            import saaaaaa.validation.validation_engine as validation_engine
            self.assertTrue(hasattr(validation_engine, 'ValidationEngine'))
            self.assertTrue(hasattr(validation_engine, 'ValidationReport'))
        except ImportError as e:
            self.fail(f"Failed to import validation_engine: {e}")

    def test_seed_factory_imports(self):
        """Verify seed_factory.py imports correctly."""
        try:
            import saaaaaa.core.seed_factory as seed_factory
            self.assertTrue(hasattr(seed_factory, 'SeedFactory'))
            self.assertTrue(hasattr(seed_factory, 'DeterministicContext'))
            self.assertTrue(hasattr(seed_factory, 'create_deterministic_seed'))
        except ImportError as e:
            self.fail(f"Failed to import seed_factory: {e}")

    def test_qmcm_hooks_imports(self):
        """Verify qmcm_hooks.py imports correctly."""
        try:
            import saaaaaa.core.qmcm_hooks as qmcm_hooks
            self.assertTrue(hasattr(qmcm_hooks, 'QMCMRecorder'))
            self.assertTrue(hasattr(qmcm_hooks, 'get_global_recorder'))
            self.assertTrue(hasattr(qmcm_hooks, 'qmcm_record'))
        except ImportError as e:
            self.fail(f"Failed to import qmcm_hooks: {e}")

    def test_evidence_registry_imports(self):
        """Verify evidence_registry.py imports correctly."""
        try:
            import saaaaaa.core.evidence_registry as evidence_registry
            self.assertTrue(hasattr(evidence_registry, 'EvidenceRegistry'))
            self.assertTrue(hasattr(evidence_registry, 'EvidenceRecord'))
        except ImportError as e:
            self.fail(f"Failed to import evidence_registry: {e}")

    def test_json_contract_loader_imports(self):
        """Verify json_contract_loader.py imports correctly."""
        try:
            import saaaaaa.core.json_contract_loader as json_contract_loader
            self.assertTrue(hasattr(json_contract_loader, 'JSONContractLoader'))
            self.assertTrue(hasattr(json_contract_loader, 'ContractDocument'))
            self.assertTrue(hasattr(json_contract_loader, 'ContractLoadReport'))
        except ImportError as e:
            self.fail(f"Failed to import json_contract_loader: {e}")

    def test_validation_predicates_imports(self):
        """Verify validation/predicates.py imports correctly."""
        try:
            from saaaaaa.utils.validation.predicates import ValidationPredicates,
ValidationResult
            self.assertIsNotNone(ValidationPredicates)
            self.assertIsNotNone(ValidationResult)
        except ImportError as e:
            self.fail(f"Failed to import validation.predicates: {e}")

    def test_golden_rule_imports(self):
        """Verify validation/golden_rule.py imports correctly."""
        try:
            from saaaaaa.utils.validation.golden_rule import GoldenRuleValidator,
GoldenRuleViolation
            self.assertIsNotNone(GoldenRuleValidator)
            self.assertIsNotNone(GoldenRuleViolation)
        except ImportError as e:
            self.fail(f"Failed to import validation.golden_rule: {e}")

    def test_meso_cluster_analysis_imports(self):
        """Verify meso_cluster_analysis.py imports correctly."""
        try:
            import saaaaaa.analysis.meso_cluster_analysis as meso_cluster_analysis
            self.assertTrue(hasattr(meso_cluster_analysis, 'analyze_policy_dispersion'))
            self.assertTrue(hasattr(meso_cluster_analysis, 'reconcile_cross_metrics'))
            self.assertTrue(hasattr(meso_cluster_analysis, 'compose_cluster_posterior'))
            self.assertTrue(hasattr(meso_cluster_analysis, 'calibrate_against_peers'))
        except ImportError as e:
            self.fail(f"Failed to import meso_cluster_analysis: {e}")

    def test_seed_factory_determinism(self):
        """Verify seed_factory produces deterministic seeds."""
        from saaaaaa.core.seed_factory import create_deterministic_seed

        # Same inputs should produce same seed
        seed1 = create_deterministic_seed("test-001", question_id="Q1", policy_area="P1")
        seed2 = create_deterministic_seed("test-001", question_id="Q1", policy_area="P1")

        self.assertEqual(seed1, seed2, "SeedFactory not producing deterministic seeds")

        # Different inputs should produce different seeds
        seed3 = create_deterministic_seed("test-002", question_id="Q1", policy_area="P1")
        self.assertNotEqual(seed1, seed3, "SeedFactory producing same seed for different
inputs")

    def test_evidence_registry_immutability(self):
        """Verify evidence_registry maintains immutability."""
        import tempfile

        from saaaaaa.core.evidence_registry import EvidenceRegistry

        # Use temporary directory for storage
        with tempfile.TemporaryDirectory() as tmpdir:
            registry = EvidenceRegistry(storage_path=Path(tmpdir) / "test_evidence.jsonl")

            # Record evidence
            registry.record_evidence(
                evidence_type="test_type_1",
                payload={"data": ["evidence1", "evidence2"]},
                source_method="test_method_1",
                parent_evidence_ids=[],
                metadata={"key": "value"}
            )

            # Verify record can be retrieved by method
            records = registry.query_by_method("test_method_1")

            self.assertEqual(len(records), 1, "Should have 1 record for test_method_1")
            self.assertEqual(records[0].source_method, "test_method_1")
            self.assertEqual(records[0].evidence_type, "test_type_1")

            # Verify record can be retrieved by type
            type_records = registry.query_by_type("test_type_1")
            self.assertEqual(len(type_records), 1, "Should have 1 record of test_type_1")

    def test_validation_engine_preconditions(self):
        """Verify validation_engine properly validates preconditions."""
        from saaaaaa.utils.validation_engine import ValidationEngine

        engine = ValidationEngine()

        # Valid preconditions
        question_spec = {
            "id": "TEST-Q1",
            "expected_elements": ["element1", "element2"]
        }
        execution_results = {"result_key": "result_value"}
        plan_text = "This is a test plan document"

        result = engine.validate_scoring_preconditions(
            question_spec, execution_results, plan_text
        )

        self.assertTrue(result.is_valid, "Valid preconditions should pass")

        # Invalid preconditions (missing expected_elements)
        invalid_spec = {"id": "TEST-Q2"}
        result2 = engine.validate_scoring_preconditions(
            invalid_spec, execution_results, plan_text
        )

        self.assertFalse(result2.is_valid, "Invalid preconditions should fail")

    def test_golden_rule_validator(self):
        """Verify golden_rule validator enforces immutability."""
        from saaaaaa.utils.validation.golden_rule import GoldenRuleValidator,
GoldenRuleViolation

        step_catalog = ["step1", "step2", "step3"]
        questionnaire_hash = "abc123"

        validator = GoldenRuleValidator(questionnaire_hash, step_catalog)

        # Should pass with same hash and catalog
        try:
            validator.assert_immutable_metadata(questionnaire_hash, step_catalog)
        except GoldenRuleViolation:
            self.fail("Golden rule should not raise for identical metadata")

        # Should fail with different hash
        with self.assertRaises(GoldenRuleViolation):
            validator.assert_immutable_metadata("different_hash", step_catalog)

        # Should fail with different catalog
        with self.assertRaises(GoldenRuleViolation):
            validator.assert_immutable_metadata(questionnaire_hash, ["step1", "step2"])

    def test_qmcm_recorder_functionality(self):
        """Verify QMCM recorder tracks method calls."""
        from saaaaaa.core.qmcm_hooks import QMCMRecorder

        recorder = QMCMRecorder()
        recorder.clear_recording()
        # Record some calls
        recorder.record_call(
            method_name="test_method",
            input_types={"arg1": "str", "arg2": "int"},
            output_type="dict",
            execution_status="success",
            execution_time_ms=10.5
        )

        stats = recorder.get_statistics()

        self.assertEqual(stats['total_calls'], 1)
        self.assertEqual(stats['unique_methods'], 1)
        self.assertEqual(stats['success_rate'], 1.0)
        self.assertEqual(stats['most_called_method'], "test_method")

    def test_json_contract_loader_functionality(self):
        """Verify JSON contract loader handles contracts correctly."""
        import json
        import tempfile

        from saaaaaa.core.json_contract_loader import JSONContractLoader

        loader = JSONContractLoader()

        # Create a temporary JSON contract
        with tempfile.NamedTemporaryFile(mode='w', suffix='.json', delete=False) as f:
            test_contract = {"key": "value", "number": 42}
            json.dump(test_contract, f)
            temp_path = f.name

        try:
            report = loader.load([temp_path])

            self.assertTrue(report.is_successful)
            self.assertEqual(len(report.documents), 1)

            doc = list(report.documents.values())[0]
            self.assertEqual(doc.payload["key"], "value")
            self.assertEqual(doc.payload["number"], 42)
            self.assertIsNotNone(doc.checksum)
        finally:
            Path(temp_path).unlink()

class TestStrategicFileInteraction(unittest.TestCase):
    """Test suite for interaction between strategic files."""

    def test_validation_engine_uses_predicates(self):
        """Verify validation_engine properly integrates with predicates."""
        from validation.predicates import ValidationPredicates
        from saaaaaa.utils.validation_engine import ValidationEngine

        engine = ValidationEngine()

        # Verify engine has predicates instance
        self.assertIsInstance(engine.predicates, ValidationPredicates)

    def test_seed_factory_context_manager(self):
        """Verify seed_factory context manager maintains state."""
        import random

        from saaaaaa.core.seed_factory import DeterministicContext

        # Save original state
        random.random()

        # Use deterministic context
        with DeterministicContext(correlation_id="test-001") as seed:
            self.assertIsInstance(seed, int)
            value_in_context = random.random()

        # Verify state is restored after context
        value_after_context = random.random()

        # Values should be different (state restored)
        self.assertNotEqual(value_in_context, value_after_context)

if __name__ == '__main__':
    unittest.main()

===== FILE: tests/test_structure_verification.py =====
#!/usr/bin/env python3
"""
Verify that the SAAAAAA repository structure is findable by Python.

This script checks:
1. All __init__.py files exist where needed
2. Python can import from all major modules
3. Package structure is correct
4. No circular import issues
"""

import sys
from pathlib import Path

from saaaaaa.config.paths import PROJECT_ROOT

repo_root = PROJECT_ROOT

def check_init_files():
    """Verify all Python packages have __init__.py files."""
    print("=" * 70)
    print("CHECKING __init__.py FILES")
    print("=" * 70)

    src_dir = repo_root / "src" / "saaaaaa"

    # Find all directories that should be packages
    package_dirs = []
    for path in src_dir.rglob("*.py"):
        if path.name != "__init__.py":
            package_dirs.append(path.parent)

    # Remove duplicates and check for __init__.py
    package_dirs = sorted(set(package_dirs))
    missing = []

    for pkg_dir in package_dirs:
        init_file = pkg_dir / "__init__.py"
        if not init_file.exists():
            missing.append(pkg_dir.relative_to(repo_root))
            print(f"✗ MISSING: {pkg_dir.relative_to(repo_root)}/__init__.py")
        else:
            print(f"✓ {pkg_dir.relative_to(repo_root)}/__init__.py")

    if missing:
        print(f"\n⚠   {len(missing)} package(s) missing __init__.py")
        return False
    else:
        print("\n✓ All packages have __init__.py")
        return True

def check_major_imports():
    """Test importing from all major modules."""
    print("\n" + "=" * 70)
    print("CHECKING MAJOR MODULE IMPORTS")
    print("=" * 70)

    imports_to_test = [
        # Core modules
        ("saaaaaa.core.orchestrator", "Orchestrator"),
        ("saaaaaa.core.orchestrator.core", "Evidence"),
        ("saaaaaa.core.orchestrator.evidence_registry", "EvidenceRegistry"),

        # Analysis modules
        ("saaaaaa.analysis.scoring.scoring", "apply_scoring"),
        ("saaaaaa.analysis.bayesian_multilevel_system", "MultiLevelBayesianOrchestrator"),

        # Processing modules
        ("saaaaaa.processing.document_ingestion", "RawDocument"),
        ("saaaaaa.processing.aggregation", "AreaPolicyAggregator"),

        # Utilities
        ("saaaaaa.utils.contracts", "validate_contract"),
        ("saaaaaa.concurrency.concurrency", "WorkerPool"),

        # Compatibility wrappers
        ("orchestrator", "Orchestrator"),
        ("scoring.scoring", "apply_scoring"),
        ("contracts", "validate_contract"),
    ]

    failed = []
    for module_name, item_name in imports_to_test:
        try:
            module = __import__(module_name, fromlist=[item_name])
            item = getattr(module, item_name)
            print(f"✓ from {module_name} import {item_name}")
        except Exception as e:
            print(f"✗ FAILED: from {module_name} import {item_name}")
            print(f"  Error: {e}")
            failed.append((module_name, item_name, str(e)))

    if failed:
        print(f"\n⚠  {len(failed)} import(s) failed")
        return False
    else:
        print("\n✓ All major imports successful")
        return True

def check_package_structure():
    """Verify the package structure is correct."""
    print("\n" + "=" * 70)
    print("CHECKING PACKAGE STRUCTURE")
    print("=" * 70)

    required_dirs = [
        "src/saaaaaa",
        "src/saaaaaa/core",
        "src/saaaaaa/core/orchestrator",
        "src/saaaaaa/analysis",
        "src/saaaaaa/analysis/scoring",
        "src/saaaaaa/processing",
        "src/saaaaaa/utils",
        "src/saaaaaa/concurrency",
    ]

    all_ok = True
    for dir_path in required_dirs:
        full_path = repo_root / dir_path
        if full_path.is_dir():
            print(f"✓ {dir_path}/")
        else:
            print(f"✗ MISSING: {dir_path}/")
            all_ok = False

    if all_ok:
        print("\n✓ Package structure is correct")
    else:
        print("\n⚠  Some directories are missing")

    return all_ok

def check_compatibility_wrappers():
    """Verify compatibility wrappers exist and work."""
    print("\n" + "=" * 70)
    print("CHECKING COMPATIBILITY WRAPPERS")
    print("=" * 70)

    wrapper_files = [
        "orchestrator.py",
        "scoring.py",
        "contracts.py",
        "aggregation.py",
        "bayesian_multilevel_system.py",
        "derek_beach.py",
        "document_ingestion.py",
        "macro_prompts.py",
        "micro_prompts.py",
        "meso_cluster_analysis.py",
    ]

    wrapper_dirs = [
        "orchestrator/",
        "scoring/",
        "concurrency/",
        "contracts/",
        "core/",
        "executors/",
    ]

    all_ok = True

    # Check wrapper files
    for wrapper in wrapper_files:
        path = repo_root / wrapper
        if path.exists():
            print(f"✓ {wrapper}")
        else:
            print(f"✗ MISSING: {wrapper}")
            all_ok = False

    # Check wrapper directories
    for wrapper in wrapper_dirs:
        path = repo_root / wrapper
        if path.is_dir() and (path / "__init__.py").exists():
            print(f"✓ {wrapper}__init__.py")
        else:
            print(f"✗ MISSING: {wrapper}__init__.py")
            all_ok = False

    if all_ok:
        print("\n✓ All compatibility wrappers present")
    else:
        print("\n⚠  Some wrappers are missing")

    return all_ok

def main():
    """Run all verification checks."""
    print("SAAAAAA REPOSITORY STRUCTURE VERIFICATION")
    print("=" * 70)
    print(f"Repository: {repo_root}")
    print()

    checks = [
        ("Package Structure", check_package_structure),
        ("__init__.py Files", check_init_files),
        ("Compatibility Wrappers", check_compatibility_wrappers),
        ("Major Imports", check_major_imports),
    ]

    results = {}
    for name, check_func in checks:
        try:
            results[name] = check_func()
        except Exception as e:
            print(f"\n✗ {name} check FAILED with exception: {e}")
            results[name] = False

    # Summary
    print("\n" + "=" * 70)
    print("SUMMARY")
    print("=" * 70)
    for name, passed in results.items():
        status = "✓ PASS" if passed else "✗ FAIL"
        print(f"{status}: {name}")

    all_passed = all(results.values())
    print("\n" + "=" * 70)
    if all_passed:
        print("✓ ALL CHECKS PASSED - Repository structure is correct!")
        return 0
    else:
        print("✕ SOME CHECKS FAILED - See details above")
        return 1

if __name__ == "__main__":
    sys.exit(main())

===== FILE: tests/test_system_audit.py =====
"""Comprehensive system audit tests - compilation, imports, routes, paths."""
from __future__ import annotations

import ast
import json
from pathlib import Path

import pytest

# Shared configuration
EXCLUDED_DIRS = {'__pycache__', '.git', 'minipdm', '.augment', '.venv'}
API_SERVER_PATHS = [
    'src/saaaaaa/api/api_server.py',
    'api/api_server.py',
    'saaaaaa/api/api_server.py',
]

class TestCompilation:
    """Test that all Python files compile successfully."""

    @pytest.fixture
    def root_path(self) -> Path:
        """Get repository root path."""
        return Path(__file__).parent.parent

    def get_python_files(self, root: Path) -> list[Path]:
        """Get all Python files in the repository."""
        files = []
        for py_file in root.rglob("*.py"):
            # Skip excluded directories
            if any(d in py_file.parts for d in EXCLUDED_DIRS):
                continue
            files.append(py_file)
        return files

    def test_all_files_compile(self, root_path: Path) -> None:
        """Test that all Python files compile without syntax errors."""
        files = self.get_python_files(root_path)
        errors = []

        for py_file in files:
            try:
                with open(py_file, encoding='utf-8') as f:
                    source = f.read()
                ast.parse(source, filename=str(py_file))
            except SyntaxError as e:
                rel_path = py_file.relative_to(root_path)
                errors.append(f"{rel_path}: SyntaxError at line {e.lineno}: {e.msg}")

        assert len(errors) == 0, f"Found {len(errors)} compilation errors:\n" +
"\n".join(errors)

    def test_no_duplicate_lines(self, root_path: Path) -> None:
        """Test that files don't have suspicious duplicate code blocks."""
        files = self.get_python_files(root_path)
        issues = []

        for py_file in files:
            try:
                with open(py_file, encoding='utf-8') as f:
                    lines = f.readlines()

                # Check for consecutive duplicate lines (more than 3)
                if len(lines) > 5:
                    for i in range(len(lines) - 3):
                        if (lines[i] == lines[i+1] == lines[i+2] == lines[i+3] and
                            len(lines[i].strip()) > 0):
                            rel_path = py_file.relative_to(root_path)
                            issues.append(f"{rel_path}: Duplicate lines at {i+1}")
                            break
            except Exception:
                pass

        assert len(issues) == 0, "Found suspicious duplicates:\n" + "\n".join(issues)

class TestImports:
    """Test import statements and module structure."""

    @pytest.fixture
    def root_path(self) -> Path:
        """Get repository root path."""
        return Path(__file__).parent.parent

    def test_no_circular_imports_in_core(self, root_path: Path) -> None:
        """Test that core modules don't have obvious circular imports."""
        # This is a basic check - full circular dependency detection requires runtime
        core_dirs = ['core', 'orchestrator', 'executors', 'concurrency']

        for dir_name in core_dirs:
            dir_path = root_path / dir_name
            if not dir_path.exists():
                continue

            for py_file in dir_path.rglob("*.py"):
                if '__pycache__' in str(py_file):
                    continue
            try:
                with open(py_file, encoding='utf-8') as f:
                    tree = ast.parse(f.read())

                # Just verify it parses - circular imports will fail at runtime
                assert tree is not None
            except SyntaxError as e:
                rel_path = py_file.relative_to(root_path)
                pytest.fail(f"Syntax error in {rel_path}: {e}")

    def test_import_statements_valid(self, root_path: Path) -> None:
        """Test that import statements are syntactically valid."""
        errors = []

        for py_file in root_path.rglob("*.py"):
            if any(d in py_file.parts for d in EXCLUDED_DIRS):
                continue

            try:
                with open(py_file, encoding='utf-8') as f:
                    tree = ast.parse(f.read())

                # Verify all import nodes are valid
                for node in ast.walk(tree):
                    if isinstance(node, ast.Import):
                        for alias in node.names:
                            assert alias.name is not None
                    elif isinstance(node, ast.ImportFrom) and node.level > 0:
                        # Relative imports should have level > 0
                        assert node.level <= 5  # Reasonable depth
            except Exception as e:
                rel_path = py_file.relative_to(root_path)
                errors.append(f"{rel_path}: {e}")

        assert len(errors) == 0, "Found import errors:\n" + "\n".join(errors)

class TestRoutes:
    """Test API routes and endpoints."""

    @pytest.fixture
    def root_path(self) -> Path:
        """Get repository root path."""
        return Path(__file__).parent.parent

    def test_api_routes_exist(self, root_path: Path) -> None:
        """Test that API server file exists and defines routes."""
        # Try to find API server file dynamically
        api_server = None
        for path in API_SERVER_PATHS:
            candidate = root_path / path
            if candidate.exists():
                api_server = candidate
                break

        if api_server is None:
            pytest.skip("API server file not found")

        with open(api_server, encoding='utf-8') as f:
            content = f.read()

        # Check for route definitions
        assert 'api/' in content, "No API routes found in api_server.py"
        assert 'health' in content or 'status' in content, "No health check endpoint
found"

class TestPaths:
    """Test file paths and references."""
    @pytest.fixture
    def root_path(self) -> Path:
        """Get repository root path."""
        return Path(__file__).parent.parent

    def test_config_files_exist(self, root_path: Path) -> None:
        """Test that expected configuration files exist."""
        expected_files = [
            'pyproject.toml',
            'requirements.txt',
            'Makefile',
        ]

        missing = []
        for file_name in expected_files:
            if not (root_path / file_name).exists():
                missing.append(file_name)

        assert len(missing) == 0, f"Missing config files: {missing}"

    def test_directory_structure(self, root_path: Path) -> None:
        """Test that expected directories exist."""
        expected_dirs = [
            'src',
            'tests',
            'core',
            'orchestrator',
        ]

        missing = []
        for dir_name in expected_dirs:
            if not (root_path / dir_name).exists():
                missing.append(dir_name)

        assert len(missing) == 0, f"Missing directories: {missing}"

    def test_audit_report_exists(self, root_path: Path) -> None:
        """Test that audit report was generated."""
        audit_report = root_path / 'docs' / 'AUDIT_REPORT.json'

        assert audit_report.exists(), "Audit report not found"

        with open(audit_report, encoding='utf-8') as f:
            data = json.load(f)

        assert 'summary' in data
        assert 'compilation_status' in data
        assert data['compilation_status'] == 'PASS'

class TestSystemIntegrity:
    """Test overall system integrity."""

    @pytest.fixture
    def root_path(self) -> Path:
        """Get repository root path."""
        return Path(__file__).parent.parent

    def test_no_empty_python_files(self, root_path: Path) -> None:
        """Test that Python files are not empty."""
        empty_files = []

        for py_file in root_path.rglob("*.py"):
            if any(d in py_file.parts for d in EXCLUDED_DIRS):
                continue

            if py_file.stat().st_size == 0:
                rel_path = py_file.relative_to(root_path)
                empty_files.append(str(rel_path))

        assert len(empty_files) == 0, f"Found empty files: {empty_files}"

    def test_init_files_present(self, root_path: Path) -> None:
        """Test that package directories have __init__.py files."""
        missing = []

        # Check key package directories
        package_dirs = [
            'core',
            'orchestrator',
            'executors',
            'concurrency',
            'scoring',
            'validation',
            'contracts',
        ]

        for dir_name in package_dirs:
            dir_path = root_path / dir_name
            if dir_path.exists() and dir_path.is_dir():
                init_file = dir_path / '__init__.py'
                if not init_file.exists():
                    missing.append(dir_name)

        # Allow some directories to not have __init__.py
        critical_missing = [d for d in missing if d not in ['tools', 'scripts']]
        assert len(critical_missing) == 0, f"Missing __init__.py in: {critical_missing}"

===== FILE: tests/test_wiring_core.py =====
"""Core wiring tests without heavy dependencies.

Tests the wiring system components in isolation without requiring
full TensorFlow/transformers imports.
"""

import json
from pathlib import Path

import pytest

from saaaaaa.core.orchestrator.signals import SignalClient, SignalRegistry
from saaaaaa.core.wiring.bootstrap import (
    CANONICAL_POLICY_AREA_DEFINITIONS,
    WiringBootstrap,
)
from saaaaaa.core.wiring.contracts import (
    CPPDeliverable,
    SPCDeliverable,
    PreprocessedDocumentDeliverable,
    SignalPackDeliverable,
)
from saaaaaa.core.wiring.errors import (
    WiringContractError,
    WiringInitializationError,
    MissingDependencyError,
)
from saaaaaa.core.wiring.feature_flags import WiringFeatureFlags
from saaaaaa.core.wiring.validation import WiringValidator


class TestWiringContracts:
    """Test contract models."""

    def test_spc_deliverable_valid(self):
        """Test valid SPC deliverable."""
        data = {
            "chunk_graph": {"chunks": {}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        spc = SPCDeliverable.model_validate(data)

        assert spc.provenance_completeness == 1.0
        assert spc.schema_version == "2.0"

    def test_spc_deliverable_invalid_provenance(self):
        """Test SPC deliverable with invalid provenance."""
        data = {
            "chunk_graph": {"chunks": {}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 0.5,  # Not 1.0!
            "schema_version": "2.0",
        }

        with pytest.raises(ValueError) as exc_info:
            SPCDeliverable.model_validate(data)

        assert "provenance_completeness" in str(exc_info.value)

    def test_cpp_deliverable_deprecated(self):
        """Test that CPPDeliverable raises deprecation warning."""
        data = {
            "chunk_graph": {"chunks": {}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        with pytest.warns(DeprecationWarning, match="CPPDeliverable is deprecated"):
            CPPDeliverable.model_validate(data)

    def test_preprocessed_document_deliverable_valid(self):
        """Test valid PreprocessedDocument deliverable."""
        data = {
            "sentence_metadata": [{"id": "s1", "text": "test"}],
            "resolution_index": {"micro": []},
            "provenance_completeness": 1.0,
            "document_id": "doc123",
        }

        doc = PreprocessedDocumentDeliverable.model_validate(data)

        assert doc.document_id == "doc123"
        assert len(doc.sentence_metadata) == 1

    def test_preprocessed_document_empty_sentences(self):
        """Test PreprocessedDocument with empty sentences."""
        data = {
            "sentence_metadata": [],  # Empty!
            "resolution_index": {},
            "provenance_completeness": 1.0,
            "document_id": "doc123",
        }

        with pytest.raises(ValueError):
            PreprocessedDocumentDeliverable.model_validate(data)

    def test_signal_pack_deliverable_valid(self):
        """Test valid SignalPack deliverable."""
        data = {
            "version": "1.0.0",
            "policy_area": "fiscal",
            "patterns": ["pattern1", "pattern2"],
            "indicators": ["ind1"],
        }

        signal = SignalPackDeliverable.model_validate(data)

        assert signal.version == "1.0.0"
        assert signal.policy_area == "fiscal"

    def test_signal_pack_empty_version(self):
        """Test SignalPack with empty version."""
        data = {
            "version": "",  # Empty!
            "policy_area": "fiscal",
            "patterns": [],
            "indicators": [],
        }

        with pytest.raises(ValueError):
            SignalPackDeliverable.model_validate(data)


class TestWiringErrors:
    """Test error classes."""

    def test_wiring_contract_error(self):
        """Test WiringContractError."""
        error = WiringContractError(
            link="cpp->adapter",
            expected_schema="CPPDeliverable",
            received_schema="dict",
            field="provenance_completeness",
            fix="Ensure ingestion completes successfully",
        )

        assert error.details["link"] == "cpp->adapter"
        assert error.details["field"] == "provenance_completeness"
        assert "Fix:" in str(error)

    def test_missing_dependency_error(self):
        """Test MissingDependencyError."""
        error = MissingDependencyError(
            dependency="questionnaire.json",
            required_by="WiringBootstrap",
            fix="Create questionnaire file",
        )

        assert error.details["dependency"] == "questionnaire.json"
        assert "Fix:" in str(error)

    def test_wiring_initialization_error(self):
        """Test WiringInitializationError."""
        error = WiringInitializationError(
            phase="load_resources",
            component="QuestionnaireResourceProvider",
            reason="File not found",
        )

        assert error.details["phase"] == "load_resources"
        assert "File not found" in str(error)


class TestWiringFeatureFlags:
    """Test feature flags."""

    def test_default_flags(self):
        """Test default flag values."""
        flags = WiringFeatureFlags()
        assert flags.use_cpp_ingestion is True
        assert flags.enable_http_signals is False
        assert flags.wiring_strict_mode is True

    def test_flags_to_dict(self):
        """Test flag serialization."""
        flags = WiringFeatureFlags(
            use_cpp_ingestion=False,
            enable_http_signals=True,
        )

        flags_dict = flags.to_dict()

        assert flags_dict["use_cpp_ingestion"] is False
        assert flags_dict["enable_http_signals"] is True

    def test_flags_validation_http_determinism_warning(self):
        """Test warning for HTTP + deterministic mode."""
        flags = WiringFeatureFlags(
            enable_http_signals=True,
            deterministic_mode=True,
        )

        warnings = flags.validate()

        assert len(warnings) > 0
        assert any("http" in w.lower() for w in warnings)

    def test_flags_validation_no_strict_mode_warning(self):
        """Test warning for disabled strict mode."""
        flags = WiringFeatureFlags(wiring_strict_mode=False)

        warnings = flags.validate()

        assert any("strict" in w.lower() for w in warnings)

    def test_flags_from_env(self, monkeypatch):
        """Test loading flags from environment."""
        monkeypatch.setenv("SAAAAAA_USE_CPP_INGESTION", "false")
        monkeypatch.setenv("SAAAAAA_ENABLE_HTTP_SIGNALS", "true")
        monkeypatch.setenv("SAAAAAA_WIRING_STRICT_MODE", "false")

        flags = WiringFeatureFlags.from_env()

        assert flags.use_cpp_ingestion is False
        assert flags.enable_http_signals is True
        assert flags.wiring_strict_mode is False


class TestWiringValidation:
    """Test validation without full bootstrap."""

    def test_validator_initialization(self):
        """Test validator initialization."""
        validator = WiringValidator()

        assert validator is not None
        metrics = validator.get_all_metrics()

        # Should have all expected links
        expected_links = [
            "cpp->adapter",
            "adapter->orchestrator",
            "orchestrator->argrouter",
            "argrouter->executors",
            "signals->registry",
            "executors->aggregate",
            "aggregate->score",
            "score->report",
        ]

        for link in expected_links:
            assert link in metrics

    def test_validate_spc_to_adapter_success(self):
        """Test successful SPC → Adapter validation."""
        validator = WiringValidator()

        spc_data = {
            "chunk_graph": {"chunks": {"c1": {}}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        # Should not raise
        validator.validate_spc_to_adapter(spc_data)

        # Check metrics
        metrics = validator.get_all_metrics()
        assert metrics["spc->adapter"]["validation_count"] == 1
        assert metrics["spc->adapter"]["failure_count"] == 0

    def test_validate_spc_to_adapter_failure(self):
        """Test failed SPC → Adapter validation."""
        validator = WiringValidator()

        bad_spc_data = {
            "chunk_graph": {},
            # Missing required fields
        }

        with pytest.raises(WiringContractError) as exc_info:
            validator.validate_spc_to_adapter(bad_spc_data)

        error = exc_info.value
        assert error.details["link"] == "spc->adapter"

        # Check metrics show failure
        metrics = validator.get_all_metrics()
        assert metrics["spc->adapter"]["failure_count"] == 1

    def test_validate_cpp_to_adapter_deprecated(self):
        """Test that validate_cpp_to_adapter raises deprecation warning."""
        validator = WiringValidator()

        cpp_data = {
            "chunk_graph": {"chunks": {"c1": {}}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        with pytest.warns(DeprecationWarning, match="validate_cpp_to_adapter is 
deprecated"):
            validator.validate_cpp_to_adapter(cpp_data)

    def test_link_hash_determinism(self):
        """Test link hashing is deterministic."""
        validator = WiringValidator()

        data = {
            "key1": "value1",
            "key2": 42,
            "nested": {"a": 1, "b": 2},
        }

        hash1 = validator.compute_link_hash("cpp->adapter", data)
        hash2 = validator.compute_link_hash("cpp->adapter", data)

        assert hash1 == hash2

    def test_link_hash_order_independence(self):
        """Test link hashing is order-independent."""
        validator = WiringValidator()

        # Same data, different key order
        data1 = {"b": 2, "a": 1, "c": 3}
        data2 = {"a": 1, "c": 3, "b": 2}

        # Use a known link name
        hash1 = validator.compute_link_hash("cpp->adapter", data1)
        hash2 = validator.compute_link_hash("cpp->adapter", data2)

        assert hash1 == hash2

    def test_validation_summary(self):
        """Test validation summary."""
        validator = WiringValidator()

        # Perform some validations
        valid_cpp = {
            "chunk_graph": {"chunks": {}},
            "policy_manifest": {},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        validator.validate_cpp_to_adapter(valid_cpp)
        validator.validate_cpp_to_adapter(valid_cpp)

        summary = validator.get_summary()

        assert summary["total_validations"] == 2
        assert summary["total_failures"] == 0
        assert summary["overall_success_rate"] == 1.0


class TestWiringObservability:
    """Test observability helpers."""

    def test_has_otel_flag(self):
        """Test HAS_OTEL flag availability."""
        from saaaaaa.core.wiring.observability import HAS_OTEL

        # Should be bool
        assert isinstance(HAS_OTEL, bool)

    def test_trace_wiring_link_context(self):
        """Test trace_wiring_link context manager."""
        from saaaaaa.core.wiring.observability import trace_wiring_link

        attrs = None
        with trace_wiring_link("test_link", document_id="doc123") as dynamic_attrs:
            attrs = dynamic_attrs
            dynamic_attrs["result"] = "success"

        # Context manager should provide dict
        assert isinstance(attrs, dict)
        assert "result" in attrs

    def test_trace_wiring_init_context(self):
        """Test trace_wiring_init context manager."""
        from saaaaaa.core.wiring.observability import trace_wiring_init

        attrs = None
        with trace_wiring_init("test_phase", component="TestComponent") as dynamic_attrs:
            attrs = dynamic_attrs
            dynamic_attrs["status"] = "complete"

        assert isinstance(attrs, dict)
        assert "status" in attrs


class TestSignalSeeding:
    """Test signal seeding wiring."""

    class _StubProvider:
        def get_patterns_for_area(self, policy_area_id: str, limit: int | None = None) ->
list[str]:
            count = limit or 4
            return [f"{policy_area_id.lower()}_pattern_{i}" for i in range(count)]

    def test_seed_signals_public_seeds_all_canonical_areas(self, tmp_path):
        """seed_signals_public should seed all canonical policy areas and aliases."""
        monolith_path = tmp_path / "monolith.json"
        monolith_path.touch()
        executor_path = tmp_path / "executor.json"
        executor_path.touch()

        bootstrap = WiringBootstrap(
            questionnaire_path=monolith_path,
            questionnaire_hash="dummy_hash",
            executor_config_path=executor_path,
            calibration_profile="default",
            abort_on_insufficient=False,
            resource_limits={},
            flags=WiringFeatureFlags()
        )
        client = SignalClient(base_url="memory://")
        registry = SignalRegistry()
        provider = self._StubProvider()

        metrics = bootstrap.seed_signals_public(client, registry, provider)

        assert metrics["canonical_areas"] == len(CANONICAL_POLICY_AREA_DEFINITIONS)
        assert metrics["hit_rate"] == pytest.approx(1.0)

        for area_id in CANONICAL_POLICY_AREA_DEFINITIONS:
            pack = registry.get(area_id)
            assert pack is not None, f"Missing signal pack for {area_id}"
            assert pack.metadata.get("canonical_id") == area_id
            assert pack.patterns, "Expected patterns for canonical signal pack"

        # Legacy aliases remain available for backward compatibility
        assert registry.get("fiscal") is not None


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_wiring_e2e.py =====
"""End-to-end tests for wiring system.

Tests cover:
- Memory mode golden document flow
- Signal unavailability graceful degradation
- ArgRouter strict mode (no silent drops)
- Contract mismatches with prescriptive errors
- Determinism (stable hashes across runs)
- Import-time budget
```

```
"""

import json
import time
from pathlib import Path

import pytest

from saaaaaa.core.wiring.bootstrap import WiringBootstrap
from saaaaaa.core.wiring.contracts import (
    AdapterExpectation,
    CPPDeliverable,
    PreprocessedDocumentDeliverable,
)
from saaaaaa.core.wiring.errors import WiringContractError
from saaaaaa.core.wiring.feature_flags import WiringFeatureFlags
from saaaaaa.core.wiring.validation import WiringValidator


@pytest.fixture
def bootstrap_args(tmp_path):
    """Provides common arguments for WiringBootstrap."""
    monolith_path = tmp_path / "monolith.json"
    monolith_path.write_text('{}')
    monolith_path.chmod(0o444)  # Read-only

    executor_path = tmp_path / "executor.json"
    executor_path.write_text('{}')
    executor_path.chmod(0o444)  # Read-only, consistent with monolith_path
    return {
        "questionnaire_path": monolith_path,
        "questionnaire_hash": "dummy_hash",
        "executor_config_path": executor_path,
        "calibration_profile": "default",
        "abort_on_insufficient": False,
        "resource_limits": {},
    }

class TestWiringBootstrap:
    """Test wiring bootstrap initialization."""

    def test_bootstrap_with_defaults(self, bootstrap_args):
        """Test bootstrap with default settings."""
        flags = WiringFeatureFlags()
        bootstrap = WiringBootstrap(**bootstrap_args, flags=flags)

        components = bootstrap.bootstrap()

        assert components is not None
        assert components.provider is not None
        assert components.signal_client is not None
        assert components.signal_registry is not None
        assert components.factory is not None
        assert components.arg_router is not None
        assert components.validator is not None
        assert components.flags == flags

    def test_bootstrap_memory_mode(self, bootstrap_args):
        """Test bootstrap in memory mode (default)."""
        flags = WiringFeatureFlags(
            use_cpp_ingestion=True,
            enable_http_signals=False,  # Memory mode
        )
        bootstrap = WiringBootstrap(**bootstrap_args, flags=flags)

        components = bootstrap.bootstrap()

        # Verify memory mode
        assert components.signal_client._transport == "memory"
        assert components.signal_client._memory_source is not None

    def test_bootstrap_with_questionnaire(self, bootstrap_args, tmp_path):
        """Test bootstrap with questionnaire file."""
        # Create temporary questionnaire
        questionnaire_data = {
            "metadata": {"version": "1.0.0"},
            "questions": [],
        }

        questionnaire_path = tmp_path / "questionnaire.json"
        questionnaire_path.write_text(json.dumps(questionnaire_data))
        questionnaire_path.chmod(0o444)

        bootstrap_args["questionnaire_path"] = questionnaire_path
        bootstrap = WiringBootstrap(**bootstrap_args)
        components = bootstrap.bootstrap()

        assert components.provider is not None

    def test_bootstrap_signals_seeded(self, bootstrap_args):
        """Test that signals are seeded in memory mode."""
        flags = WiringFeatureFlags(enable_http_signals=False)
        bootstrap = WiringBootstrap(**bootstrap_args, flags=flags)

        components = bootstrap.bootstrap()

        # Check that registry has some signals
        metrics = components.signal_registry.get_metrics()
        assert metrics["size"] > 0, "Registry should have seeded signals"

    def test_bootstrap_argrouter_coverage(self, bootstrap_args):
        """Test that ArgRouter has required route coverage."""
        bootstrap = WiringBootstrap(**bootstrap_args)
        components = bootstrap.bootstrap()

        coverage = components.arg_router.get_special_route_coverage()

        assert coverage >= 30, f"Expected ≥30 special routes, got {coverage}"



class TestWiringValidation:
    """Test contract validation between links."""

    def test_spc_to_adapter_valid(self):
        """Test valid SPC → Adapter contract."""
        validator = WiringValidator()

        spc_data = {
            "chunk_graph": {"chunks": {"chunk1": {}}},
            "policy_manifest": {"version": "1.0.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0.0",
        }

        # Should not raise
        validator.validate_spc_to_adapter(spc_data)

    def test_spc_to_adapter_missing_provenance(self):
        """Test SPC → Adapter with incomplete provenance."""
        validator = WiringValidator()

        spc_data = {
            "chunk_graph": {"chunks": {"chunk1": {}}},
            "policy_manifest": {"version": "1.0.0"},
            "provenance_completeness": 0.5,  # Not 1.0!
            "schema_version": "2.0.0",
        }

        with pytest.raises(WiringContractError) as exc_info:
            validator.validate_spc_to_adapter(spc_data)

        error = exc_info.value
        assert "provenance_completeness" in str(error).lower()
        assert error.details["link"] == "spc->adapter"

    def test_adapter_to_orchestrator_valid(self):
        """Test valid Adapter → Orchestrator contract."""
        validator = WiringValidator()

        doc_data = {
            "sentence_metadata": [{"id": "sent1", "text": "test"}],
            "resolution_index": {"micro": []},
            "provenance_completeness": 1.0,
            "document_id": "doc123",
        }

        # Should not raise
        validator.validate_adapter_to_orchestrator(doc_data)

    def test_adapter_to_orchestrator_empty_sentences(self):
        """Test Adapter → Orchestrator with empty sentences."""
        validator = WiringValidator()

        doc_data = {
            "sentence_metadata": [],  # Empty!
            "resolution_index": {"micro": []},
            "provenance_completeness": 1.0,
            "document_id": "doc123",
        }

        with pytest.raises(WiringContractError) as exc_info:
            validator.validate_adapter_to_orchestrator(doc_data)

        error = exc_info.value
        assert "sentence_metadata" in str(error).lower()

    def test_orchestrator_to_argrouter_valid(self):
        """Test valid Orchestrator → ArgRouter contract."""
        validator = WiringValidator()

        payload_data = {
            "class_name": "TestAnalyzer",
            "method_name": "_extract_claims",
            "payload": {"content": "test"},
        }

        # Should not raise
        validator.validate_orchestrator_to_argrouter(payload_data)

    def test_signals_to_registry_valid(self):
        """Test valid Signals → Registry contract."""
        validator = WiringValidator()

        signal_data = {
            "version": "1.0.0",
            "policy_area": "fiscal",
            "patterns": ["pattern1", "pattern2"],
            "indicators": [],
        }

        # Should not raise
        validator.validate_signals_to_registry(signal_data)
    def test_signals_to_registry_missing_version(self):
        """Test Signals → Registry with missing version."""
        validator = WiringValidator()

        signal_data = {
            "version": "",  # Empty version!
            "policy_area": "fiscal",
            "patterns": [],
            "indicators": [],
        }

        with pytest.raises(WiringContractError) as exc_info:
            validator.validate_signals_to_registry(signal_data)

        error = exc_info.value
        assert "version" in str(error).lower()

    def test_link_hash_determinism(self):
        """Test that link hashes are deterministic."""
        validator = WiringValidator()

        data = {
            "key1": "value1",
            "key2": 42,
            "key3": ["a", "b", "c"],
        }

        hash1 = validator.compute_link_hash("cpp->adapter", data)
        hash2 = validator.compute_link_hash("cpp->adapter", data)

        assert hash1 == hash2, "Hashes should be deterministic"

    def test_validation_metrics(self):
        """Test that validation metrics are tracked."""
        validator = WiringValidator()

        # Perform some validations
        spc_data = {
            "chunk_graph": {"chunks": {}},
            "policy_manifest": {},
            "provenance_completeness": 1.0,
            "schema_version": "2.0.0",
        }

        validator.validate_spc_to_adapter(spc_data)
        validator.validate_spc_to_adapter(spc_data)

        metrics = validator.get_all_metrics()

        assert "spc->adapter" in metrics
        assert metrics["spc->adapter"]["validation_count"] == 2
        assert metrics["spc->adapter"]["failure_count"] == 0


class TestWiringDeterminism:
    """Test determinism guarantees."""


class TestWiringFeatureFlags:
    """Test feature flag functionality."""

    def test_default_flags(self):
        """Test default flag values."""
        flags = WiringFeatureFlags()

        assert flags.use_cpp_ingestion is True
        assert flags.enable_http_signals is False
        assert flags.allow_threshold_override is False
        assert flags.wiring_strict_mode is True

    def test_flags_to_dict(self):
        """Test flag serialization."""
        flags = WiringFeatureFlags()

        flags_dict = flags.to_dict()

        assert isinstance(flags_dict, dict)
        assert "use_cpp_ingestion" in flags_dict
        assert "enable_http_signals" in flags_dict

    def test_flags_validation_warnings(self):
        """Test flag validation warnings."""
        # HTTP + deterministic should warn
        flags = WiringFeatureFlags(
            enable_http_signals=True,
            deterministic_mode=True,
        )

        warnings = flags.validate()

        assert len(warnings) > 0
        assert any("http" in w.lower() for w in warnings)

    def test_flags_from_env(self, monkeypatch):
        """Test loading flags from environment."""
        monkeypatch.setenv("SAAAAAA_USE_CPP_INGESTION", "false")
        monkeypatch.setenv("SAAAAAA_ENABLE_HTTP_SIGNALS", "true")

        flags = WiringFeatureFlags.from_env()

        assert flags.use_cpp_ingestion is False
        assert flags.enable_http_signals is True


class TestWiringImportTime:
    """Test import time performance."""

    def test_import_time_budget(self):
        """Test that wiring imports complete within budget."""
        start = time.time()

        # Import all wiring modules
        from saaaaaa.core.wiring import bootstrap, contracts, errors, feature_flags,
observability, validation

        elapsed = time.time() - start

        # Should complete in under 1 second
        assert elapsed < 1.0, f"Import took {elapsed:.2f}s, exceeds 1s budget"


class TestWiringE2EGoldenFlow:
    """Test end-to-end golden document flow."""

    def test_golden_flow_memory_mode(self, bootstrap_args):
        """Test complete flow in memory mode."""
        # Bootstrap system
        flags = WiringFeatureFlags(
            use_cpp_ingestion=True,
            enable_http_signals=False,
            deterministic_mode=True,
        )

        bootstrap = WiringBootstrap(**bootstrap_args, flags=flags)
        components = bootstrap.bootstrap()
        # Verify all components initialized
        assert components.provider is not None
        assert components.signal_registry is not None
        assert components.factory is not None
        assert components.arg_router is not None

        # Verify signals available
        registry_metrics = components.signal_registry.get_metrics()
        assert registry_metrics["hit_rate"] >= 0.0  # May be 0 if no fetches yet
        assert registry_metrics["size"] > 0  # Should have seeded signals

        # Verify router coverage
        route_coverage = components.arg_router.get_special_route_coverage()
        assert route_coverage >= 30

        # Verify validator ready
        validator_summary = components.validator.get_summary()
        assert validator_summary["overall_success_rate"] == 1.0  # No failures yet

    def test_golden_flow_with_validation(self, bootstrap_args):
        """Test flow with contract validation at each step."""
        bootstrap = WiringBootstrap(**bootstrap_args)
        components = bootstrap.bootstrap()

        validator = components.validator

        # Simulate CPP → Adapter
        cpp_data = {
            "chunk_graph": {"chunks": {"c1": {}}},
            "policy_manifest": {"version": "1.0"},
            "provenance_completeness": 1.0,
            "schema_version": "2.0",
        }

        validator.validate_cpp_to_adapter(cpp_data)

        # Simulate Adapter → Orchestrator
        doc_data = {
            "sentence_metadata": [{"id": "s1"}],
            "resolution_index": {},
            "provenance_completeness": 1.0,
            "document_id": "doc1",
        }

        validator.validate_adapter_to_orchestrator(doc_data)

        # Check metrics
        metrics = validator.get_all_metrics()
        assert metrics["cpp->adapter"]["validation_count"] == 1
        assert metrics["adapter->orchestrator"]["validation_count"] == 1


class TestWiringErrorHandling:
    """Test error handling and prescriptive messages."""

    def test_contract_error_has_fix(self):
        """Test that contract errors include fix instructions."""
        validator = WiringValidator()

        bad_spc_data = {
            "chunk_graph": {},  # Missing required fields
        }

        try:
            validator.validate_spc_to_adapter(bad_spc_data)
            pytest.fail("Should have raised WiringContractError")
        except WiringContractError as e:
            # Check error has details
            assert e.details is not None
            assert "link" in e.details
            assert e.details["link"] == "spc->adapter"

            # Check error message is prescriptive
            error_msg = str(e)
            assert "spc->adapter" in error_msg.lower()

    def test_initialization_error_prescriptive(self):
        """Test that initialization errors are prescriptive."""
        from saaaaaa.core.wiring.errors import WiringInitializationError

        error = WiringInitializationError(
            phase="load_resources",
            component="QuestionnaireResourceProvider",
            reason="File not found",
        )

        error_msg = str(error)
        assert "load_resources" in error_msg
        assert "QuestionnaireResourceProvider" in error_msg
        assert "File not found" in error_msg


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/unit/__init__.py =====

===== FILE: tests/validate_spc_implementation.py =====
#!/usr/bin/env python3
"""
Standalone validation script for SPC exploitation features.

This script validates the chunk-aware functionality without requiring pytest.
It can be run directly to verify the implementation.
"""

import sys
from pathlib import Path

# Add src to path
REPO_ROOT = Path(__file__).parent.parent


def test_imports():
    """Test that all required modules can be imported."""
    print("Testing imports...")

    try:
        from saaaaaa.core.orchestrator.core import ChunkData, PreprocessedDocument
        print("✓ ChunkData and PreprocessedDocument imported")
    except ImportError as e:
        print(f"✗ Failed to import core types: {e}")
        return False

    try:
        from saaaaaa.core.orchestrator.chunk_router import ChunkRouter, ChunkRoute
        print("✓ ChunkRouter imported")
    except ImportError as e:
        print(f"✗ Failed to import ChunkRouter: {e}")
        return False

    try:
        from saaaaaa.processing.cpp_ingestion.models import CanonPolicyPackage, Chunk
        print("✓ CPP models imported")
    except ImportError as e:
        print(f"✗ Failed to import CPP models: {e}")
        return False

    try:
        from saaaaaa.analysis.spc_causal_bridge import SPCCausalBridge
        print("✓ SPCCausalBridge imported")
    except ImportError as e:
        print(f"✗ Failed to import SPCCausalBridge: {e}")
        return False

    return True


def test_chunk_data_creation():
    """Test creating ChunkData objects."""
    print("\nTesting ChunkData creation...")

    try:
        from saaaaaa.core.orchestrator.core import ChunkData

        chunk = ChunkData(
            id=1,
            text="Test chunk",
            chunk_type="diagnostic",
            sentences=[0, 1],
            tables=[],
            start_pos=0,
            end_pos=50,
            confidence=0.9,
            edges_out=[2],
            edges_in=[0],
        )

        assert chunk.id == 1
        assert chunk.chunk_type == "diagnostic"
        assert chunk.confidence == 0.9
        print("✓ ChunkData created successfully")
        return True
    except Exception as e:
        print(f"✗ ChunkData creation failed: {e}")
        return False


def test_chunk_router():
    """Test ChunkRouter functionality."""
    print("\nTesting ChunkRouter...")

    try:
        from saaaaaa.core.orchestrator.core import ChunkData
        from saaaaaa.core.orchestrator.chunk_router import ChunkRouter

        router = ChunkRouter()

        # Test all chunk types
        chunk_types = ["diagnostic", "activity", "indicator", "resource", "temporal",
"entity"]

        for chunk_type in chunk_types:
            chunk = ChunkData(
                id=0,
                text="test",
                chunk_type=chunk_type,
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
            )
            route = router.route_chunk(chunk)

            if route.executor_class == "":
                print(f"✗ No executor for chunk type: {chunk_type}")
                return False

            print(f"  ✓ {chunk_type} → {route.executor_class}")

        print("✓ ChunkRouter routing successful")
        return True
    except Exception as e:
        print(f"✗ ChunkRouter test failed: {e}")
        import traceback
        traceback.print_exc()
        return False


def test_preprocessed_document():
    """Test PreprocessedDocument with chunks."""
    print("\nTesting PreprocessedDocument...")

    try:
        from saaaaaa.core.orchestrator.core import ChunkData, PreprocessedDocument

        # Test chunked mode
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[i],
                tables=[],
                start_pos=i*10,
                end_pos=(i+1)*10,
                confidence=0.9,
            )
            for i in range(3)
        ]

        doc = PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test text",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            chunk_index={},
            chunk_graph={},
            processing_mode="chunked",
        )

        assert doc.processing_mode == "chunked"
        assert len(doc.chunks) == 3
        print("✓ PreprocessedDocument in chunked mode works")

        # Test flat mode (backward compatibility)
        doc_flat = PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test text",
            sentences=[],
            tables=[],
            metadata={},
        )

        assert doc_flat.processing_mode == "flat"
        assert len(doc_flat.chunks) == 0
        print("✓ PreprocessedDocument in flat mode works (backward compatible)")
        return True
    except Exception as e:
        print(f"✗ PreprocessedDocument test failed: {e}")
        import traceback
        traceback.print_exc()
        return False


def test_spc_causal_bridge():
    """Test SPCCausalBridge functionality."""
    print("\nTesting SPCCausalBridge...")

    try:
        from saaaaaa.analysis.spc_causal_bridge import SPCCausalBridge

        bridge = SPCCausalBridge()

        # Test causal weight mapping
        assert bridge._compute_causal_weight("dependency") > 0.8
        assert bridge._compute_causal_weight("sequential") < 0.5
        assert bridge._compute_causal_weight("unknown") == 0.0
        print("✓ Causal weight mapping correct")

        # Test graph building (if networkx available)
        try:
            import networkx as nx

            chunk_graph = {
                "nodes": [
                    {"id": 0, "type": "diagnostic", "text": "Node 0", "confidence": 0.9},
                    {"id": 1, "type": "activity", "text": "Node 1", "confidence": 0.8},
                ],
                "edges": [
                    {"source": 0, "target": 1, "type": "sequential"},
                ],
            }

            G = bridge.build_causal_graph_from_spc(chunk_graph)

            if G is not None:
                assert G.number_of_nodes() == 2
                assert G.number_of_edges() == 1
                print("✓ Causal graph building successful")
            else:
                print("⚠ NetworkX not available, skipping graph test")
        except ImportError:
            print("⚠ NetworkX not available, skipping graph test")

        return True
    except Exception as e:
        print(f"✗ SPCCausalBridge test failed: {e}")
        import traceback
        traceback.print_exc()
        return False


def main():
    """Run all validation tests."""
    print("=" * 80)
    print("SPC EXPLOITATION VALIDATION TESTS")
    print("=" * 80)

    tests = [
        ("Imports", test_imports),
        ("ChunkData Creation", test_chunk_data_creation),
        ("ChunkRouter", test_chunk_router),
        ("PreprocessedDocument", test_preprocessed_document),
        ("SPCCausalBridge", test_spc_causal_bridge),
    ]

    results = []
    for name, test_func in tests:
        try:
            result = test_func()
            results.append((name, result))
        except Exception as e:
            print(f"\n✗ {name} crashed: {e}")
            import traceback
            traceback.print_exc()
            results.append((name, False))

    print("\n" + "=" * 80)
    print("SUMMARY")
    print("=" * 80)

    passed = sum(1 for _, result in results if result)
    total = len(results)

    for name, result in results:
        status = "✓ PASS" if result else "✗ FAIL"
        print(f"{status}: {name}")

    print("=" * 80)
    print(f"Results: {passed}/{total} tests passed")
    print("=" * 80)

    return passed == total


if __name__ == "__main__":
    success = main()
    sys.exit(0 if success else 1)
```

===== FILE: tests/verify_anchoring.py =====

```
"""Verify Method Anchoring.

Checks that all methods in the codebase are anchored to the central calibration system
via @calibrated_method or direct usage of the orchestrator.
"""

import ast
import os
import sys

def verify_all_methods_anchored():
    """
    OBLIGATORY: Script that verifies ALL methods are anchored.
    """

    errors = []
    repo_root = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
    src_path = os.path.join(repo_root, "src/saaaaaa")

    print(f"Scanning {src_path}...")

    # 1. Scan all files
    for root, dirs, files in os.walk(src_path):
        for file in files:
            if not file.endswith(".py"):
                continue

            filepath = os.path.join(root, file)

            with open(filepath, 'r') as f:
                try:
```

```
                tree = ast.parse(f.read())
            except Exception as e:
                print(f"Skipping {file}: {e}")
                continue

        # 2. Search methods
        for node in ast.walk(tree):
            if not isinstance(node, ast.FunctionDef):
                continue

            # Ignore private and special methods
            if node.name.startswith("_"):
                continue

            # 3. Verify @calibrated_method decorator
            has_calibrated_decorator = any(
                isinstance(dec, ast.Call) and
                getattr(dec.func, 'id', None) == 'calibrated_method'
                for dec in node.decorator_list
            )

            # 4. Or uses orchestrator/param_loader in body
            uses_orchestrator = False
            uses_param_loader = False

            for child in ast.walk(node):
                if isinstance(child, ast.Name):
                    if 'orchestrator' in child.id.lower():
                        uses_orchestrator = True
                    if 'param' in child.id.lower() and 'loader' in child.id.lower():
                        uses_param_loader = True

            # 5. If NEITHER -> ERROR
            if not (has_calibrated_decorator or uses_orchestrator or
uses_param_loader):
                # Check for hardcoded values (heuristic)
                has_hardcoded = False

                for child in ast.walk(node):
                    if isinstance(child, ast.Constant) and isinstance(child.value,
(int, float)):
                         if 0.0 <= child.value <= 1.0:
                            has_hardcoded = True
                            break
                    # Support older python versions where Num is used
                    elif isinstance(child, ast.Num):
                        if 0.0 <= child.n <= 1.0:
                            has_hardcoded = True
                            break

                if has_hardcoded:
                    errors.append({
                        "file": filepath,
                        "method": node.name,
                        "line": node.lineno,
                        "error": "Method has hardcoded values but is not anchored to
central system"
                    })

    # 6. REPORT
    if errors:
        print("❌ FOUND UNANCHORED METHODS:")
        for error in errors:
            print(f"  {error['file']}:{error['line']} - {error['method']}")
            print(f"    → {error['error']}")

        # In a real CI, we would raise AssertionError.
        # For this task, we print and maybe fail if strict.
        # raise AssertionError(f"Found {len(errors)} unanchored methods.")
        return False

    print(f"✓ All methods properly anchored to central system")
    return True

if __name__ == "__main__":
    success = verify_all_methods_anchored()
    if not success:
        sys.exit(1)

===== FILE: tmp/tmpxdn7ddfb/_remote_module_non_scriptable.py =====
from typing import *

import torch
import torch.distributed.rpc as rpc
from torch import Tensor
from torch._jit_internal import Future
from torch.distributed.rpc import RRef
from typing import Tuple  # pyre-ignore: unused import


module_interface_cls = None


def forward_async(self, *args, **kwargs):
    args = (self.module_rref, self.device, self.is_device_map_set, *args)
    kwargs = {**kwargs}
    return rpc.rpc_async(
        self.module_rref.owner(),
        _remote_forward,
        args,
        kwargs,
    )


def forward(self, *args, **kwargs):
    args = (self.module_rref, self.device, self.is_device_map_set, *args)
    kwargs = {**kwargs}
    ret_fut = rpc.rpc_async(
        self.module_rref.owner(),
        _remote_forward,
        args,
        kwargs,
    )
    return ret_fut.wait()


_generated_methods = [
    forward_async,
    forward,
]




def _remote_forward(
    module_rref: RRef[module_interface_cls], device: str, is_device_map_set: bool, *args,
**kwargs):
    module = module_rref.local_value()
    device = torch.device(device)

    if device.type != "cuda":
        return module.forward(*args, **kwargs)

    # If the module is on a cuda device,
    # move any CPU tensor in args or kwargs to the same cuda device.
    # Since torch script does not support generator expression,
    # have to use concatenation instead of
    # ``tuple(i.to(device) if isinstance(i, Tensor) else i for i in *args)``.
    args = (*args,)
    out_args: Tuple[()] = ()
    for arg in args:
        arg = (arg.to(device),) if isinstance(arg, Tensor) else (arg,)
        out_args = out_args + arg

    kwargs = {**kwargs}
    for k, v in kwargs.items():
        if isinstance(v, Tensor):
            kwargs[k] = kwargs[k].to(device)

    if is_device_map_set:
        return module.forward(*out_args, **kwargs)

    # If the device map is empty, then only CPU tensors are allowed to send over wire,
    # so have to move any GPU tensor to CPU in the output.
    # Since torch script does not support generator expression,
    # have to use concatenation instead of
    # ``tuple(i.cpu() if isinstance(i, Tensor) else i for i in module.forward(*out_args,
**kwargs))``.
    ret: Tuple[()] = ()
    for i in module.forward(*out_args, **kwargs):
        i = (i.cpu(),) if isinstance(i, Tensor) else (i,)
        ret = ret + i
    return ret

===== FILE: tools/__init__.py =====
"""Tools package for utilities and scripts."""

===== FILE: tools/bulk_import_test.py =====
#!/usr/bin/env python3
"""
Bulk import test - verifies all modules can be imported
"""
import importlib
import sys
from pathlib import Path

def main() -> None:
    """Test importing all modules in the package."""
    errors = []
    success = []

    # Add src to path
    # Find all Python modules
    src_path = Path('src/saaaaaa')
    if not src_path.exists():
        print(f"Error: {src_path} does not exist")
        sys.exit(1)

    for py_file in src_path.rglob('*.py'):
        if py_file.name == '__init__.py':
            continue

        # Convert path to module name
        rel_path = py_file.relative_to('src')
        module_name = str(rel_path.with_suffix('')).replace('/', '.')

        try:
            importlib.import_module(module_name)
            success.append(module_name)
            print(f'✓ {module_name}')
        except Exception as e:
            errors.append((module_name, str(e)))
            print(f'✗ {module_name}: {e}')
    print('\n=== Bulk Import Results ===')
    print(f'Success: {len(success)} modules')
    print(f'Errors: {len(errors)} modules')

    if errors:
        print('\nFailed imports:')
        for module, error in errors[:10]:  # Show first 10
            print(f'  - {module}: {error[:100]}')
        if len(errors) > 10:
            print(f'  ... and {len(errors) - 10} more')
        sys.exit(1)
    else:
        print('✓ All modules imported successfully')
        sys.exit(0)

if __name__ == '__main__':
    main()

===== FILE: tools/detect_cycles.py =====
#!/usr/bin/env python3
"""
Simple circular dependency detector
Alternative to pycycle for detecting import cycles.
"""
import ast
import sys
from collections import defaultdict
from pathlib import Path

def extract_imports(file_path: Path) -> set[str]:
    """Extract import statements from a Python file."""
    imports = set()

    try:
        with open(file_path, encoding='utf-8') as f:
            tree = ast.parse(f.read(), str(file_path))

        for node in ast.walk(tree):
            if isinstance(node, ast.Import):
                for alias in node.names:
                    imports.add(alias.name.split('.')[0])
            elif isinstance(node, ast.ImportFrom) and node.module:
                imports.add(node.module.split('.')[0])
    except (SyntaxError, UnicodeDecodeError) as e:
        print(f"Warning: Could not parse {file_path}: {e}", file=sys.stderr)

    return imports

def build_dependency_graph(root_path: Path, package_name: str) -> dict[str, set[str]]:
    """Build a dependency graph for the package."""
    graph = defaultdict(set)

    for py_file in root_path.rglob('*.py'):
        if py_file.name == '__init__.py':
            continue

        # Convert file path to module name
        try:
            rel_path = py_file.relative_to(root_path.parent)
            module = str(rel_path.with_suffix('')).replace('/', '.')
        except ValueError:
            continue

        # Extract imports from this module
        imports = extract_imports(py_file)

        # Track all imports (internal and external) for dependency analysis
        for imp in imports:
            # Add to graph if it's an internal module or starts with package name
            graph[module].add(imp)

            # Also track sub-package relationships
            if '.' in imp and imp.startswith(package_name):
                parts = imp.split('.')
                for i in range(1, len(parts)):
                    sub_pkg = '.'.join(parts[:i+1])
                    graph[module].add(sub_pkg)

    return graph

def find_cycles(graph: dict[str, set[str]]) -> list[list[str]]:
    """Find cycles in the dependency graph using DFS."""
    cycles = []
    visited = set()
    rec_stack = set()
    path = []

    def dfs(node: str) -> bool:
        """DFS to detect cycles."""
        visited.add(node)
        rec_stack.add(node)
        path.append(node)

        for neighbor in graph.get(node, set()):
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in rec_stack:
                # Found a cycle
                cycle_start = path.index(neighbor)
                cycle = path[cycle_start:] + [neighbor]
                cycles.append(cycle)
                return True

        path.pop()
        rec_stack.remove(node)
        return False

    for node in graph:
        if node not in visited:
            dfs(node)

    return cycles

def main() -> None:
    """Main entry point."""
    if len(sys.argv) < 2:
        print("Usage: python detect_cycles.py <package_path>")
        sys.exit(1)

    root_path = Path(sys.argv[1])
    if not root_path.exists():
        print(f"Error: Path {root_path} does not exist")
        sys.exit(1)

    # Get package name from path
    package_name = root_path.name

    print(f"Analyzing package: {package_name}")
    print(f"Path: {root_path}")

    # Build dependency graph
    graph = build_dependency_graph(root_path, package_name)

    print(f"\nFound {len(graph)} modules")
    # Find cycles
    cycles = find_cycles(graph)

    if cycles:
        print(f"\n✗ Found {len(cycles)} circular dependencies:")
        for i, cycle in enumerate(cycles, 1):
            print(f"\n  Cycle {i}:")
            for j, module in enumerate(cycle):
                if j < len(cycle) - 1:
                    print(f"    {module} →")
                else:
                    print(f"    {module}")
        sys.exit(1)
    else:
        print("\n✓ No circular dependencies found")
        sys.exit(0)

if __name__ == '__main__':
    main()

===== FILE: tools/grep_boundary_checks.py =====
"""Grep-based boundary violation detector for architectural guardrails."""
from __future__ import annotations

import re
import subprocess
import sys
from pathlib import Path
from typing import TYPE_CHECKING

try:
    from saaaaaa.config.paths import PROJECT_ROOT
except Exception:  # pragma: no cover - bootstrap fallback
    PROJECT_ROOT = Path(__file__).resolve().parents[1]

if TYPE_CHECKING:
    from collections.abc import Sequence

REPO_ROOT = PROJECT_ROOT

class BoundaryViolation(Exception):
    """Raised when a boundary violation is detected."""


def run_grep(pattern: str, paths: Sequence[str]) -> list[str]:
    """Run grep command and return matching lines."""
    try:
        cmd = ["grep", "-rn", "--include=*.py", pattern, *paths]
        result = subprocess.run(
            cmd,
            cwd=REPO_ROOT,
            capture_output=True,
            text=True,
            check=False,
        )
        if result.returncode == 0:
            return result.stdout.strip().split("\n")
        return []
    except Exception as exc:
        print(f"Warning: grep command failed: {exc}", file=sys.stderr)
        return []


def check_no_orchestrator_imports_in_core() -> None:
    """Ensure core and executors don't import from orchestrator."""
    print("Checking: core/executors must not import orchestrator...")

    patterns = [
        r"import\s+orchestrator",
        r"from\s+orchestrator\s+import",
    ]

    violations = []
    for pattern in patterns:
        matches = run_grep(pattern, ["core", "executors"])
        if matches and matches != [""]:
            violations.extend(matches)

    if violations:
        print(f"✕ Found {len(violations)} orchestrator import violations in
core/executors:")
        for violation in violations[:10]:  # Show first 10
            print(f"  {violation}")
        raise BoundaryViolation("core/executors must not import orchestrator")

    print("  ✓ No orchestrator imports in core/executors")


def check_no_provider_calls_in_core() -> None:
    """Ensure core doesn't call orchestrator provider functions."""
    print("Checking: core must not call get_questionnaire_provider...")

    pattern = r"get_questionnaire_provider\s*\("
    matches = run_grep(pattern, ["core", "executors"])

    if matches and matches != [""]:
        print(f"✕ Found {len(matches)} provider calls in core/executors:")
        for match in matches[:10]:
            print(f"  {match}")
        raise BoundaryViolation("core/executors must not call orchestrator providers")

    print("  ✓ No provider calls in core/executors")


def check_no_json_io_in_core() -> None:
    """Ensure core doesn't perform direct JSON file I/O."""
    print("Checking: core must not perform JSON file I/O...")

    # More specific pattern: open() with .json inside parentheses
    pattern = r'open\([^)]*\.json[^)]*\)'
    matches = run_grep(pattern, ["core"])

    if matches and matches != [""]:
        print(f"✕ Found {len(matches)} JSON I/O operations in core:")
        for match in matches[:10]:
            print(f"  {match}")
        raise BoundaryViolation("core must not perform direct JSON I/O")

    print("  ✓ No JSON I/O in core")


def main() -> None:
    """Run all boundary checks."""
    print("=== Grep-based Boundary Checks ===\n")

    checks = [
        check_no_orchestrator_imports_in_core,
        check_no_provider_calls_in_core,
        check_no_json_io_in_core,
    ]

    failed = []
    for check in checks:
        try:
            check()
        except BoundaryViolation as exc:
            failed.append(str(exc))

    if failed:
        print(f"\n ✗ {len(failed)} boundary check(s) failed")
        sys.exit(1)

    print("\n✓ All grep-based boundary checks passed")


if __name__ == "__main__":
    main()

===== FILE: tools/integrity/__init__.py =====
"""Integrity checking tools."""

===== FILE: tools/lint/check_pythonpath_references.py =====
#!/usr/bin/env python3
"""
Fail CI when new PYTHONPATH snippets appear outside the documented allowlist.
"""

from __future__ import annotations

import sys
from pathlib import Path

ALLOWLIST = {
    Path("BUILD_HYGIENE.md"),
    Path("tools/validation/validate_build_hygiene.py"),
    Path("docs/QUICKSTART.md"),
    Path("TEST_AUDIT_REPORT.md"),
    Path("tests/conftest.py"),
    Path("STRATEGIC_WIRING_ARCHITECTURE.md"),
    Path("tools/lint/check_pythonpath_references.py"),
}


def main() -> int:
    repo_root = Path(__file__).resolve().parents[2]
    violations: list[str] = []

    for path in repo_root.rglob("*"):
        if path.suffix in {".pyc", ".png", ".jpg", ".jpeg"}:
            continue
        if any(part in {".git", ".venv", "venv", "__pycache__"} for part in path.parts):
            continue
        if not path.is_file():
            continue

        try:
            text = path.read_text(encoding="utf-8")
        except UnicodeDecodeError:
            continue

        if "PYTHONPATH" not in text:
            continue

        rel_path = path.relative_to(repo_root)

        if rel_path in ALLOWLIST:
            continue

        for idx, line in enumerate(text.splitlines(), 1):
            if "PYTHONPATH" in line:
                violations.append(f"{rel_path}:{idx}: {line.strip()}")

    if violations:
        print(" ✗ New PYTHONPATH references detected outside the allowlist:")
        for violation in violations:
            print(f"  - {violation}")
        print("\nUpdate the documentation to remove these references or extend the
allowlist intentionally.")
        return 1

    print("✓ No unexpected PYTHONPATH references found.")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())

===== FILE: tools/migrations/__init__.py =====
"""Migration tools."""

===== FILE: tools/migrations/migrate_ids_v1_to_v2.py =====
#!/usr/bin/env python3
"""Migrate legacy policy analysis configuration to atomic ID v2 format.

This script converts the historical questionnaire.json and rubric_scoring.json
files that used P#/D#/Q# identifiers into the new canonical data contracts that
use PAxx/DIMxx/Qxxx identifiers. It also normalises structure, injects i18n
metadata, and computes deterministic content hashes for reproducible builds.

Usage
-----
python tools/migrations/migrate_ids_v1_to_v2.py \
    --questionnaire questionnaire.json \
    --rubric rubric_scoring.json \
    --execution-mapping execution_mapping.yaml \
    --write  # Persist changes (omit for dry-run)
"""

from __future__ import annotations

import argparse
import copy
import datetime as dt
import hashlib
import json
import sys
from collections import Counter, defaultdict
from decimal import ROUND_HALF_UP, Decimal
from pathlib import Path
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from collections.abc import Mapping

ROOT = Path(__file__).resolve().parents[2]

LEGACY_POLICY_AREAS = [f"P{i}" for i in range(1, 11)]
LEGACY_DIMENSIONS = [f"D{i}" for i in range(1, 7)]

POLICY_AREA_IDS = [f"PA{i:02d}" for i in range(1, 11)]
DIMENSION_IDS = [f"DIM{i:02d}" for i in range(1, 7)]

SOURCES_CATALOG = [
    {
        "key": "official_stats",
        "type": "primary",
        "format": "stat",
        "auth_level": "official",
        "description": "Series estadísticas oficiales desagregadas por sexo, edad y
territorio.",
    },
    {
        "key": "official_documents",
        "type": "primary",
        "format": "narrative",
        "auth_level": "official",
        "description": "Actos administrativos, planes sectoriales y acuerdos municipales
vigentes.",
    },
    {
        "key": "monitoring_tables",
        "type": "primary",
        "format": "table",
        "auth_level": "official",
        "description": "Tableros de seguimiento con metas, responsables y ejecución
presupuestal.",
    },
    {
        "key": "third_party_research",
        "type": "secondary",
        "format": "narrative",
        "auth_level": "third_party",
        "description": "Investigaciones académicas y evaluaciones externas con evidencia
empírica.",
    },
    {
        "key": "geo_maps",
        "type": "secondary",
        "format": "map",
        "auth_level": "official",
        "description": "Capas geoespaciales oficiales (IGAC, IDEAM, UNGRD) con proyección
EPSG:3116.",
    },
]

POLICY_AREA_EVIDENCE_KEYS = {
    "PA01": ["official_stats", "official_documents", "third_party_research"],
    "PA02": ["official_stats", "monitoring_tables", "official_documents"],
    "PA03": ["official_stats", "official_documents", "geo_maps"],
    "PA04": ["official_stats", "geo_maps", "official_documents"],
    "PA05": ["official_stats", "monitoring_tables", "third_party_research"],
    "PA06": ["official_documents", "third_party_research", "monitoring_tables"],
    "PA07": ["official_stats", "official_documents", "monitoring_tables"],
    "PA08": ["geo_maps", "official_documents", "official_stats"],
    "PA09": ["official_stats", "official_documents", "monitoring_tables"],
    "PA10": ["official_stats", "third_party_research", "official_documents"],
}

ALL_EVIDENCE_KEYS = sorted({key for keys in POLICY_AREA_EVIDENCE_KEYS.values() for key in
keys})

DEFAULT_MODALITY_NA_RULES = {
    "TYPE_A": {"imputation": "mean", "scope": "dimension", "exclude_from_global": False},
    "TYPE_B": {"imputation": "mean", "scope": "dimension", "exclude_from_global": False},
    "TYPE_C": {"imputation": "median", "scope": "dimension", "exclude_from_global":
False},
    "TYPE_D": {"imputation": "zero", "scope": "dimension", "exclude_from_global": False},
    "TYPE_E": {"imputation": "none", "scope": "policy_area", "exclude_from_global":
False},
    "TYPE_F": {"imputation": "none", "scope": "policy_area", "exclude_from_global": True},
}

ROUNDING_RULES = {
    "question": {"mode": "half_up", "precision": 2},
    "dimension": {"mode": "half_up", "precision": 2},
    "policy_area": {"mode": "half_up", "precision": 1},
    "cluster": {"mode": "half_up", "precision": 1},
    "macro": {"mode": "half_up", "precision": 1},
}
```

```
UNCERTAINTY_POLICY = {
    "method": "bootstrap",
    "alpha": 0.05,
    "propagation": "weighted",
    "iterations": 2000,
}

PENALTY_RULES = {
    "contradictory_info": {"weight": 0.06},
    "missing_indicator": {"weight": 0.04},
    "OOD_flag": {"weight": 0.05},
}

IMBALANCE_POLICY = {
    "method": "gini",
    "gini_formula": "2*sum(i*xi)/(n*sum(xi)) - (n+1)/n",
    "std_dev_formula": "sqrt(sum((xi - mean)^2)/n)",
    "range_formula": "max(xi) - min(xi)",
    "thresholds": {
        "CL01": {"gini": 0.3, "std_dev": 12.0, "range": 35.0, "action": "penalize"},
        "CL02": {"gini": 0.28, "std_dev": 10.0, "range": 32.0, "action": "penalize"},
        "CL03": {"gini": 0.27, "std_dev": 9.0, "range": 28.0, "action": "flag"},
        "CL04": {"gini": 0.29, "std_dev": 11.0, "range": 30.0, "action": "penalize"},
    },
}

RECOMMENDATION_RULES = [
    {
        "cluster_id": "CL01",
        "condition": {"type": "low_score", "threshold": 60},
        "action": {
            "problem": "Déficit crítico en capacidades de seguridad y justicia local.",
            "intervention": "Implementar comités interinstitucionales con trazabilidad
presupuestal.",
            "indicator": "Porcentaje de denuncias judicializadas con atención integral.",
            "owner": "Secretaría de Seguridad",
            "timeframe": "2025-Q2",
        },
    },
    {
        "cluster_id": "CL02",
        "condition": {"type": "high_imbalance", "threshold": 0.3},
        "action": {
            "problem": "Desbalance en enfoque diferencial y coberturas para población
vulnerable.",
            "intervention": "Reasignar recursos a programas de inclusión con metas
específicas.",
            "indicator": "Número de hogares con atención integral en rutas
diferenciales.",
            "owner": "Secretaría de Desarrollo Social",
            "timeframe": "2025-Q3",
        },
    },
    {
        "cluster_id": "CL03",
        "condition": {"type": "combined", "threshold": 65},
        "action": {
            "problem": "Resultados ambientales inconsistentes con metas de mitigación.",
            "intervention": "Activar mesa técnica intersectorial para control de
deforestación.",
            "indicator": "Hectáreas restauradas con monitoreo IDEAM.",
            "owner": "Secretaría de Ambiente",
            "timeframe": "2025-Q4",
        },
    },
    {
        "cluster_id": "CL04",
        "condition": {"type": "low_score", "threshold": 55},
        "action": {
            "problem": "Protección social insuficiente ante choques migratorios y
sanitarios.",
            "intervention": "Implementar plan de contingencia con red hospitalaria y
ayudas humanitarias.",
            "indicator": "Tiempo promedio de respuesta ante emergencias sociales.",
            "owner": "Secretaría de Salud",
            "timeframe": "2025-Q1",
        },
    },
]

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--questionnaire", type=Path, default=ROOT / "questionnaire.json")
    parser.add_argument("--rubric", type=Path, default=ROOT / "rubric_scoring.json")
    parser.add_argument(
        "--execution-mapping",
        type=Path,
        default=ROOT / "execution_mapping.yaml",
        dest="execution_mapping",
    )
    parser.add_argument("--write", action="store_true", help="Persist migrated artifacts")
    parser.add_argument("--output-dir", type=Path, default=None, help="Optional output
directory")
    return parser.parse_args()

def load_json(path: Path) -> dict[str, Any]:
    with path.open("r", encoding="utf-8") as fh:
        return json.load(fh)

def canonical_hash(payload: Mapping[str, Any]) -> str:
    serialisable = json.loads(json.dumps(payload, ensure_ascii=False))
    serialisable = dict(serialisable)
    serialisable.pop("content_hash", None)
    canonical = json.dumps(serialisable, ensure_ascii=False, sort_keys=True,
separators=(",", ":"))
    return hashlib.sha256(canonical.encode("utf-8")).hexdigest()

def dump_json(path: Path, payload: Mapping[str, Any]) -> None:
    payload = copy.deepcopy(payload)
    payload["content_hash"] = canonical_hash(payload)
    with path.open("w", encoding="utf-8") as fh:
        json.dump(payload, fh, ensure_ascii=False, indent=2, sort_keys=True)
        fh.write("\n")

def map_policy_area(legacy_id: str) -> str:
    index = LEGACY_POLICY_AREAS.index(legacy_id)
    return POLICY_AREA_IDS[index]

def map_dimension(legacy_id: str) -> str:
    index = LEGACY_DIMENSIONS.index(legacy_id)
    return DIMENSION_IDS[index]

def decimal_normalised(value: float, digits: int = 6) -> float:
    quant = Decimal(value).quantize(Decimal(10) ** -digits, rounding=ROUND_HALF_UP)
    return float(quant)

def build_i18n(label_es: str, label_en: str | None = None) -> dict[str, Any]:
    if label_en is None:
        label_en = label_es
    return {
        "default": "es",
        "keys": {
            "label_es": label_es,
            "label_en": label_en,
        },
    }
```

```
def migrate_questionnaire(
    questionnaire: dict[str, Any],
    legacy_modalities: Mapping[str, str] | None = None,
    legacy_policy_area_labels: Mapping[str, str] | None = None,
) -> tuple[dict[str, Any], dict[str, str], dict[str, str], dict[str, list[str]]]:
    metadata = questionnaire.get("metadata", {})
    legacy_clusters = metadata.get("clusters", [])
    legacy_dimensions = questionnaire.get("dimensiones", {})
    questions = questionnaire.get("preguntas_base", [])

    pa_mapping = {legacy: map_policy_area(legacy) for legacy in LEGACY_POLICY_AREAS}
    dim_mapping = {legacy: map_dimension(legacy) for legacy in LEGACY_DIMENSIONS}

    cluster_by_pa: dict[str, str] = {}
    clusters: list[dict[str, Any]] = []
    for cluster in legacy_clusters:
        cluster_id = cluster["cluster_id"]
        pa_ids = []
        for raw in cluster.get("policy_area_ids", []):
            if raw in pa_mapping:
                pa_ids.append(pa_mapping[raw])
            elif raw in POLICY_AREA_IDS:
                pa_ids.append(raw)
        if not pa_ids and cluster.get("legacy_point_ids"):
            for raw in cluster.get("legacy_point_ids", []):
                if raw in pa_mapping:
                    pa_ids.append(pa_mapping[raw])
        for pa_id in pa_ids:
            cluster_by_pa[pa_id] = cluster_id
        legacy_ids = [raw for raw in cluster.get("legacy_point_ids", []) if raw in
pa_mapping]
        if not legacy_ids:
            legacy_ids = [raw for raw in cluster.get("policy_area_ids", []) if raw in
LEGACY_POLICY_AREAS]

        clusters.append(
            {
                "cluster_id": cluster_id,
                "i18n": build_i18n(cluster["name"], cluster["name"]),
                "rationale": cluster.get("rationale", ""),
                "policy_area_ids": pa_ids,
                "legacy_policy_area_ids": legacy_ids,
            }
        )

    policy_area_names: dict[str, str] = dict(legacy_policy_area_labels or {})
    for question in questions:
        pa_name = question.get("policy_area_name")
        legacy_pa = question.get("id", "").split("-")[0]
        if legacy_pa and pa_name and legacy_pa not in policy_area_names:
            policy_area_names[legacy_pa] = pa_name

    policy_area_entries: list[dict[str, Any]] = []
    for legacy_id in LEGACY_POLICY_AREAS:
        policy_area_id = pa_mapping[legacy_id]
        policy_area_entries.append(
            {
                "policy_area_id": policy_area_id,
                "legacy_ids": [legacy_id],
                "i18n": build_i18n(policy_area_names.get(legacy_id, legacy_id)),
                "cluster_id": cluster_by_pa.get(policy_area_id, ""),
                "dimension_ids": DIMENSION_IDS,
                "required_evidence_keys": POLICY_AREA_EVIDENCE_KEYS[policy_area_id],
            }
        )

    dimension_entries: list[dict[str, Any]] = []
    for legacy_dim, dim_payload in legacy_dimensions.items():
        dim_id = dim_mapping[legacy_dim]
        dimension_entries.append(
            {
                "dimension_id": dim_id,
                "legacy_id": legacy_dim,
                "i18n": build_i18n(dim_payload.get("nombre", legacy_dim)),
                "description": dim_payload.get("descripcion", ""),
            }
        )

    question_entries: list[dict[str, Any]] = []
    legacy_to_new_qid: dict[str, str] = {}
    policy_area_sequence: dict[str, int] = defaultdict(int)
    combination_sequence: dict[tuple[str, str], int] = defaultdict(int)

    questions_sorted = sorted(questions, key=lambda q: q.get("id", ""))
    for index, question in enumerate(questions_sorted, start=1):
        legacy_id = question.get("id")
        if not legacy_id:
            continue
        legacy_pa, legacy_dim, legacy_q = legacy_id.split("-")
        policy_area_id = pa_mapping[legacy_pa]
        dimension_id = dim_mapping[legacy_dim]
        cluster_id = cluster_by_pa.get(policy_area_id, "")

        global_order = index
        policy_area_sequence[policy_area_id] += 1
        combination_key = (policy_area_id, dimension_id)
        combination_sequence[combination_key] += 1

        question_id = f"Q{index:03d}"
        legacy_to_new_qid[legacy_id] = question_id

        modality = "TYPE_A"
        if legacy_modalities and legacy_id in legacy_modalities:
            modality = legacy_modalities[legacy_id]

        question_entries.append(
            {
                "question_id": question_id,
                "legacy_id": legacy_id,
                "policy_area_id": policy_area_id,
                "dimension_id": dimension_id,
                "cluster_id": cluster_id,
                "order": {
                    "global": global_order,
                    "within_policy_area": policy_area_sequence[policy_area_id],
                    "within_policy_area_dimension": combination_sequence[combination_key],
                },
                "question_text": question.get("texto_template", ""),
                "i18n": build_i18n(question.get("texto_template", "")),
                "scoring_modality": modality,
                "required_evidence_keys": POLICY_AREA_EVIDENCE_KEYS[policy_area_id],
                "evidence_expectations": question.get("criterios_evaluacion", {}),
                "search_patterns": {
                    "regex": question.get("patrones_verificacion", []),
                },
                "scoring_criteria": question.get("scoring", {}),
                "validation_checks": question.get("verificacion_lineas_base", {}),
            }
        )

    title = metadata.get("title", "Configuración de 300 Preguntas - Sistema de Evaluación
Causal FARFAN 3.0")
    description = metadata.get("description", "")

    migrated = {
        "version": "3.0.0",
        "provenance": {
            "author": "Data Contracts Team",
            "tool": "migrate_ids_v1_to_v2.py",
            "edited_at":
dt.datetime.now(dt.timezone.utc).replace(microsecond=0).isoformat(),
        },
        "changelog": [
            {
                "version": "3.0.0",
                "summary": "Migración a identificadores atómicos y estructura
normalizada",
                "reason": "Data contract hardening",
            }
        ],
        "metadata": {
            "default_language": "es",
            "title": build_i18n(title, "Policy Evaluation Questionnaire"),
            "description": description,
            "policy_areas": policy_area_entries,
            "dimensions": dimension_entries,
            "clusters": clusters,
            "sources_of_verification": SOURCES_CATALOG,
        },
        "questions": question_entries,
    }

    return migrated, legacy_to_new_qid, pa_mapping, cluster_by_pa

def weight_vector_from_mapping(mapping: Mapping[str, Any], pa_mapping: Mapping[str, str])
-> dict[str, dict[str, float]]:
    weights_by_pa: dict[str, dict[str, float]] = {}
    for legacy_dim, payload in mapping.items():
        dim_id = map_dimension(legacy_dim)
        deca_map = payload.get("decalogo_dimension_mapping", {})
        for legacy_pa, details in deca_map.items():
            pa_id = pa_mapping[legacy_pa]
            weights_by_pa.setdefault(pa_id, {})[dim_id] =
decimal_normalised(details.get("weight", 0.0))
    return {pa_id: normalise_weights(dim_weights) for pa_id, dim_weights in
weights_by_pa.items()}

def normalise_weights(weights: Mapping[str, float]) -> dict[str, float]:
    total = sum(weights.values())
    if not total:
        return dict.fromkeys(weights, 0.0)
    return {k: decimal_normalised(v / total) for k, v in weights.items()}

def migrate_rubric(
    rubric: dict[str, Any],
    legacy_questionnaire: dict[str, Any],
    migrated_questionnaire: dict[str, Any],
    legacy_to_new_qid: Mapping[str, str],
    pa_mapping: Mapping[str, str],
    _cluster_by_pa: Mapping[str, str],
) -> dict[str, Any]:
    legacy_dimensions = legacy_questionnaire.get("dimensiones", {})
    modality_definitions = rubric.get("scoring_modalities", {})
    legacy_questions = rubric.get("questions", [])
    cluster_weights = rubric.get("meso_clusters", {})
    aggregation_levels = rubric.get("aggregation_levels", {})

    question_modality: dict[str, str] = {}
    for question in legacy_questions:
        legacy_id = question.get("id")
        modality = question.get("scoring_modality", "TYPE_A")
        if legacy_id and legacy_id in legacy_to_new_qid:
            question_modality[legacy_to_new_qid[legacy_id]] = modality
    # Build rubric matrix with allowed modalities per PA/DIM
    modality_counter: dict[tuple[str, str], Counter] = defaultdict(Counter)
    for legacy_id, new_qid in legacy_to_new_qid.items():
        pa_legacy, dim_legacy, _ = legacy_id.split("-")
        pa_id = pa_mapping[pa_legacy]
        dim_id = map_dimension(dim_legacy)
        modality = question_modality.get(new_qid, "TYPE_A")
        modality_counter[(pa_id, dim_id)][modality] += 1

    rubric_matrix: dict[str, dict[str, Any]] = {}
    for pa_id in POLICY_AREA_IDS:
        rubric_matrix[pa_id] = {}
        for dim_id in DIMENSION_IDS:
            counter = modality_counter.get((pa_id, dim_id), Counter({"TYPE_A": 1}))
            allowed = sorted(counter.keys())
            default_modality = max(counter.items(), key=lambda kv: kv[1])[0]
            rubric_matrix[pa_id][dim_id] = {
                "default_modality": default_modality,
                "allowed_modalities": allowed,
                "required_evidence_keys": POLICY_AREA_EVIDENCE_KEYS[pa_id],
            }

    # Build dimension -> question weights
    dimension_question_weights: dict[str, dict[str, float]] = {dim: {} for dim in
DIMENSION_IDS}
    for question in migrated_questionnaire["questions"]:
        dim_id = question["dimension_id"]
        qid = question["question_id"]
        dimension_question_weights[dim_id].setdefault(qid, 0.0)

    for _dim_id, questions_dict in dimension_question_weights.items():
        count = len(questions_dict)
        if not count:
            continue
        weight = decimal_normalised(1 / count)
        for qid in list(questions_dict.keys()):
            questions_dict[qid] = weight

    # Policy area -> dimension weights
    policy_area_dimension_weights = weight_vector_from_mapping(legacy_dimensions,
pa_mapping)

    # Cluster weights (policy areas within cluster)
    cluster_policy_area_weights: dict[str, dict[str, float]] = {}
    for cluster_id, payload in cluster_weights.items():
        pa_weights = {pa_mapping[pa]: weight for pa, weight in payload.get("weights",
{}).items()}
        cluster_policy_area_weights[cluster_id] = {
            pa_id: decimal_normalised(weight) for pa_id, weight in
normalise_weights(pa_weights).items()
        }

    macro_weights = aggregation_levels.get("level_4", {}).get("cluster_weights", {})
    macro_cluster_weights = {cluster: decimal_normalised(weight) for cluster, weight in
normalise_weights(macro_weights).items()}

    # Scoring modalities enriched metadata
    scoring_modalities: dict[str, Any] = {}
    for modality_id, payload in modality_definitions.items():
        enriched = {
            "name": payload.get("id", modality_id),
            "description": payload.get("description", ""),
            "score_range": {
                "min": 0.0,
                "max": payload.get("max_score", 3.0),
            },
            "rounding": ROUNDING_RULES["question"],
            "required_evidence_keys": ALL_EVIDENCE_KEYS,
        }
        if "expected_elements" in payload:
            enriched["expected_elements"] = payload["expected_elements"]
        if "conversion_table" in payload:
            enriched["conversion_table"] = payload["conversion_table"]
        if payload.get("uses_semantic_matching"):
            enriched["determinism"] = {"seed_required": True, "seed_source":
"seed_factory_v1"}
        scoring_modalities[modality_id] = enriched

    rubric_payload = {
        "version": "3.0.0",
        "requires_questionnaire_version": "3.0.0",
        "provenance": {
            "author": "Data Contracts Team",
            "tool": "migrate_ids_v1_to_v2.py",
            "edited_at":
dt.datetime.now(dt.timezone.utc).replace(microsecond=0).isoformat(),
        },
        "changelog": [
            {
                "version": "3.0.0",
                "summary": "Migración de rúbrica a esquema con pesos únicos e imputación
NA",
                "reason": "Data contract hardening",
            }
        ],
        "metadata": {
            "default_language": "es",
            "rounding": ROUNDING_RULES,
            "uncertainty": UNCERTAINTY_POLICY,
            "tie_breaker_notes": "Los puntajes exactos en los límites superiores (p. ej.
84.5) se asignan a la banda superior tras redondeo half_up.",
        },
        "scoring_modalities": scoring_modalities,
        "na_rules": {
            "modalities": DEFAULT_MODALITY_NA_RULES,
            "policy_areas": {
                pa_id: {
                    "imputation": ("mean" if pa_id not in {"PA03", "PA08"} else "median"),
                    "scope": "policy_area",
                    "exclude_from_global": pa_id in {"PA03", "PA08"},
                }
                for pa_id in POLICY_AREA_IDS
            },
        },
        "penalties": PENALTY_RULES,
        "aggregation": {
            "dimension_question_weights": dimension_question_weights,
            "policy_area_dimension_weights": policy_area_dimension_weights,
            "cluster_policy_area_weights": cluster_policy_area_weights,
            "macro_cluster_weights": macro_cluster_weights,
        },
        "score_bands": rubric.get("score_bands", {}),
        "rubric_matrix": rubric_matrix,
        "recommendation_rules": RECOMMENDATION_RULES,
        "imbalance": IMBALANCE_POLICY,
        "uncertainty": UNCERTAINTY_POLICY,
        "required_evidence_keys": POLICY_AREA_EVIDENCE_KEYS,
    }

    return rubric_payload

def update_metadata_checksums(questionnaire_path: Path, rubric_path: Path,
execution_mapping_path: Path, output_path: Path) -> dict[str, str]:
    questionnaire_payload = load_json(questionnaire_path)
    rubric_payload = load_json(rubric_path)
    def canonical_yaml_hash_from_file(path: Path) -> str:
        raw = path.read_text(encoding="utf-8")
        normalised_lines = [line.rstrip() for line in raw.splitlines()]
        normalised_text = "\n".join(normalised_lines).strip() + "\n"
        return hashlib.sha256(normalised_text.encode("utf-8")).hexdigest()

    checksums = {
        "questionnaire.json": canonical_hash(questionnaire_payload),
        "rubric_scoring.json": canonical_hash(rubric_payload),
        "execution_mapping.yaml": canonical_yaml_hash_from_file(execution_mapping_path),
    }

    output_path.parent.mkdir(parents=True, exist_ok=True)
    with output_path.open("w", encoding="utf-8") as fh:
        json.dump(checksums, fh, ensure_ascii=False, indent=2, sort_keys=True)
        fh.write("\n")
    return checksums

def main() -> None:
    args = parse_args()

    questionnaire = load_json(args.questionnaire)
    rubric = load_json(args.rubric)

    legacy_modalities = {
        item.get("id"): item.get("scoring_modality", "TYPE_A")
        for item in rubric.get("questions", [])
        if item.get("id")
    }

    legacy_policy_area_labels: dict[str, str] = {}
    for item in rubric.get("questions", []):
        legacy_id = item.get("id")
        if not legacy_id:
            continue
        pa_legacy = legacy_id.split("-")[0]
        if pa_legacy not in legacy_policy_area_labels and item.get("policy_area_name"):
            legacy_policy_area_labels[pa_legacy] = item["policy_area_name"]

    migrated_questionnaire, legacy_to_new_qid, pa_mapping, cluster_by_pa = 
migrate_questionnaire(
        questionnaire,
        legacy_modalities,
        legacy_policy_area_labels,
    )
    migrated_rubric = migrate_rubric(
        rubric,
        questionnaire,
        migrated_questionnaire,
        legacy_to_new_qid,
        pa_mapping,
        cluster_by_pa,
    )

    output_dir = args.output_dir or args.questionnaire.parent
    questionnaire_target = output_dir / args.questionnaire.name
    rubric_target = output_dir / args.rubric.name

    if args.write:
        dump_json(questionnaire_target, migrated_questionnaire)
        dump_json(rubric_target, migrated_rubric)
        update_metadata_checksums(
            questionnaire_target,
            rubric_target,
            args.execution_mapping,
            ROOT / "config" / "metadata_checksums.json",
        )
    else:
        preview = {
            "questionnaire_sample": migrated_questionnaire["questions"][0],
            "rubric_matrix_sample": migrated_rubric["rubric_matrix"]["PA01"]["DIM01"],
        }
        json.dump(preview, fp=sys.stdout, ensure_ascii=False, indent=2)

if __name__ == "__main__":
    main()

===== FILE: tools/orchestration_condition_audit.py =====
#!/usr/bin/env python3
"""
Advanced orchestration readiness evaluator.

This script performs SOTA-style detection of objective, necessary, and sufficient
conditions required for orchestrator implementation. It validates runtime
constraints, critical file presence, phase wiring, router guarantees, registry
integrity, and questionnaire resource extraction guarantees, and produces a JSON
report consumable by runbooks or CI instrumentation.
"""

from __future__ import annotations

import inspect
import json
import sys
import threading
from pathlib import Path
from typing import Any

try:
    from saaaaaa.config.paths import PROJECT_ROOT
except ImportError as exc:  # pragma: no cover - configuration error
    raise SystemExit(
        "Unable to import 'saaaaaa'. Install the package with 'pip install -e .' before
running this audit."
    ) from exc

REPO_ROOT = PROJECT_ROOT
SRC_ROOT = PROJECT_ROOT / "src"


def _record(name: str, passed: bool, severity: str, details: dict[str, Any]) -> dict[str,
Any]:
    return {
        "check": name,
        "passed": passed,
        "severity": severity,
        "details": details,
    }


def check_python_version() -> dict[str, Any]:
    required_min = (3, 12)
    required_max = (3, 13)
    actual = sys.version_info[:2]
    passed = required_min <= actual < required_max
    return _record(
        "python_version_window",
        passed,
        "critical",
        {
            "required_range": f"{required_min[0]}.{required_min[1]} <= version <
{required_max[0]}.{required_max[1]}",
            "detected": f"{actual[0]}.{actual[1]}",
        },
    )
```

```
def check_critical_files() -> dict[str, Any]:
    critical_paths = [
        "src/saaaaaa/core/orchestrator/core.py",
        "src/saaaaaa/core/orchestrator/arg_router.py",
        "src/saaaaaa/core/orchestrator/class_registry.py",
        "src/saaaaaa/core/orchestrator/executors.py",
        "src/saaaaaa/core/orchestrator/factory.py",
        "src/saaaaaa/core/orchestrator/questionnaire_resource_provider.py",
        "src/saaaaaa/core/orchestrator/questionnaire.py",
        "src/saaaaaa/processing/cpp_ingestion/__init__.py",
        "src/saaaaaa/processing/cpp_ingestion/models.py",
    ]
    missing = [p for p in critical_paths if not (REPO_ROOT / p).exists()]
    return _record(
        "critical_orchestration_files_present",
        not missing,
        "critical",
        {"missing": missing, "checked": len(critical_paths)},
    )


def check_phase_definitions() -> dict[str, Any]:
    try:
        from saaaaaa.core.orchestrator.core import Orchestrator
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "orchestrator_phase_integrity",
            False,
            "critical",
            {"error": f"Unable to import Orchestrator: {exc!r}"},
        )

    phases = getattr(Orchestrator, "FASES", [])
    ids = [phase[0] for phase in phases]
    handlers_missing: list[str] = []
    mode_mismatches: list[tuple[str, str]] = []

    for _, mode, handler_name, _ in phases:
        handler = getattr(Orchestrator, handler_name, None)
        if handler is None:
            handlers_missing.append(handler_name)
            continue
        is_async = inspect.iscoroutinefunction(handler)
        if mode == "async" and not is_async:
            mode_mismatches.append((handler_name, "expected async"))
        if mode == "sync" and is_async:
            mode_mismatches.append((handler_name, "expected sync"))

    duplicate_ids = len(ids) != len(set(ids))
    monotonic = ids == sorted(ids)

    passed = bool(phases) and not handlers_missing and not mode_mismatches and not
duplicate_ids and monotonic
    return _record(
        "phase_definition_consistency",
        passed,
        "critical",
        {
            "phase_count": len(phases),
            "missing_handlers": handlers_missing,
            "mode_mismatches": mode_mismatches,
            "duplicate_ids": duplicate_ids,
            "monotonic_ids": monotonic,
        },
    )
```

```
def check_class_registry() -> dict[str, Any]:
    try:
        from saaaaaa.core.orchestrator.class_registry import build_class_registry
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "class_registry_build",
            False,
            "critical",
            {"error": f"Unable to import class registry: {exc!r}"},
        )

    try:
        registry = build_class_registry()
        registry_count = len(registry)
        sample = sorted(registry.keys())[:5]
        passed = registry_count >= 20
        return _record(
            "class_registry_build",
            passed,
            "high",
            {"count": registry_count, "sample": sample},
        )
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "class_registry_build",
            False,
            "critical",
            {"error": f"build_class_registry failed: {exc!r}"},
        )


def check_extended_router() -> dict[str, Any]:
    try:
        from saaaaaa.core.orchestrator.arg_router import ExtendedArgRouter
        from saaaaaa.core.orchestrator.class_registry import build_class_registry
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "extended_router_integrity",
            False,
            "high",
            {"error": f"Unable to import router dependencies: {exc!r}"},
        )

    try:
        router = ExtendedArgRouter(build_class_registry())
    except Exception as exc:
        return _record(
            "extended_router_integrity",
            False,
            "high",
            {"error": f"ExtendedArgRouter initialization failed: {exc!r}"},
        )

    lock_ok = isinstance(getattr(router, "_lock", None), threading.RLock)
    special_routes = getattr(router, "_special_routes", {})
    route_count = len(special_routes)
    passed = lock_ok and route_count >= 25
    return _record(
        "extended_router_integrity",
        passed,
        "medium",
        {
            "lock_is_rlock": lock_ok,
            "special_route_count": route_count,
        },
    )
```

```
def check_questionnaire_provider() -> dict[str, Any]:
    try:
        from saaaaaa.core.orchestrator.questionnaire_resource_provider import
QuestionnaireResourceProvider
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "questionnaire_provider_extracts",
            False,
            "medium",
            {"error": f"Unable to import QuestionnaireResourceProvider: {exc!r}"},
        )

    sample_questionnaire = {
        "version": "diagnostic",
        "schema_version": "diagnostic",
        "blocks": {
            "micro_questions": [],
            "meso_questions": [],
            "macro_question": {},
        },
        "validations": [],
    }

    provider = QuestionnaireResourceProvider(sample_questionnaire)
    patterns = provider.extract_all_patterns()
    validations = provider.extract_all_validations()
    passed = isinstance(patterns, list) and isinstance(validations, list)
    return _record(
        "questionnaire_provider_extracts",
        passed,
        "medium",
        {
            "pattern_count": len(patterns),
            "validation_count": len(validations),
        },
    )


def check_executor_registry() -> dict[str, Any]:
    try:
        from saaaaaa.core.orchestrator.core import Orchestrator
    except Exception as exc:  # pragma: no cover - defensive
        return _record(
            "executor_registry_coverage",
            False,
            "high",
            {"error": f"Unable to import Orchestrator: {exc!r}"},
        )

    executors = getattr(Orchestrator, "executors", {})
    passed = isinstance(executors, dict) and len(executors) >= 25
    return _record(
        "executor_registry_coverage",
        passed,
        "medium",
        {"executor_count": len(executors)},
    )


def run_checks() -> dict[str, Any]:
    checks = [
        check_python_version(),
        check_critical_files(),
        check_phase_definitions(),
        check_class_registry(),
        check_extended_router(),
        check_executor_registry(),
        check_questionnaire_provider(),
    ]
    passed = sum(1 for c in checks if c["passed"])
    failed = len(checks) - passed
    return {
        "total_checks": len(checks),
        "passed": passed,
        "failed": failed,
        "results": checks,
    }


def main() -> None:
    report = run_checks()
    print(json.dumps(report, indent=2))
    if report["failed"]:
        sys.exit(1)


if __name__ == "__main__":
    main()

===== FILE: tools/prompt_cross_analysis.py =====
"""Prompt Cross analytics utilities.

This module consolidates registry information across micro, meso, and macro
levels and generates cross-cutting diagnostics for coverage, contract health,
and causal path integrity. The calculations use the synthetic dataset stored in
``data/prompt_cross_registry.json`` to demonstrate how the metrics are derived.
"""

from __future__ import annotations

import json
from collections import defaultdict
from dataclasses import dataclass
from pathlib import Path
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from collections.abc import Iterable

DATA_PATH = Path("data/prompt_cross_registry.json")

def _load_data() -> dict[str, object]:
    """Load the consolidated prompt-cross dataset."""

    with DATA_PATH.open("r", encoding="utf-8") as handle:
        return json.load(handle)

def _contribution(weight: float, normalized_time: float, depth: int) -> float:
    """Compute contribution score for a single registry entry."""

    safe_depth = max(depth, 1)
    return weight * normalized_time / safe_depth

def consolidate_evidence(records: Iterable[dict[str, object]]) -> dict[str, object]:
    """Deduplicate registry records and compute contribution metrics.

    Args:
        records: Iterable of QMCM registry records.

    Returns:
        Dictionary with consolidated nodes, deduplication ratio, and top
        contributors ranked by contribution score.
    """

    records = list(records)
```

```
canonical: dict[tuple[str, str, str], dict[str, object]] = {}
record_to_node: dict[str, str] = {}
parent_links: dict[str, list[str]] = defaultdict(list)
contributions: dict[str, float] = defaultdict(float)

for record in records:
    key = (
        record["question_id"],
        record["method_id"],
        record["hash_output"],
    )
    node_id = (
        f"{record['level']}|{record['question_id']}|"
        f"{record['method_id']}|{record['hash_output']}"
    )

    if key not in canonical:
        canonical[key] = {
            "node_id": node_id,
            "level": record["level"],
            "question_id": record["question_id"],
            "method_id": record["method_id"],
            "hash_output": record["hash_output"],
            "dimensions": set(),
            "records": [],
        }

    node_entry = canonical[key]
    node_entry["records"].append(record["record_id"])

    dimension = record.get("dimension")
    if dimension:
        node_entry["dimensions"].add(dimension)

    record_to_node[record["record_id"]] = node_id
    contributions[node_id] += _contribution(
        record["weight"], record["normalized_time"], int(record["depth"])
    )

    parent_record = record.get("parent_record")
    if parent_record:
        parent_links[node_id].append(parent_record)

# Resolve parent pointers to canonical node identifiers
resolved_parents: dict[str, list[str]] = {}
for node_id, parents in parent_links.items():
    resolved = {
        record_to_node[parent]
        for parent in parents
        if parent in record_to_node
    }
    resolved_parents[node_id] = sorted(resolved)

# Build global node list
level_order = {"micro": 0, "meso": 1, "macro": 2}
global_nodes: list[dict[str, object]] = []
for entry in canonical.values():
    node_id = entry["node_id"]
    global_nodes.append(
        {
            "node_id": node_id,
            "level": entry["level"],
            "question_id": entry["question_id"],
            "method_id": entry["method_id"],
            "hash_output": entry["hash_output"],
            "parent_nodes": resolved_parents.get(node_id, []),
            "record_count": len(entry["records"]),
            "dimensions": sorted(entry["dimensions"]),
                "contribution_score": round(contributions[node_id], 6),
            }
        )

    global_nodes.sort(
        key=lambda node: (
            level_order.get(str(node["level"]), 99),
            str(node["question_id"]),
            str(node["method_id"]),
        )
    )

    total_records = len(records)

    unique_nodes = len(global_nodes)
    dedup_ratio = unique_nodes / total_records if total_records else 0.0

    top_contributors = sorted(
        (
            {
                "node_id": node["node_id"],
                "question_id": node["question_id"],
                "method_id": node["method_id"],
                "contribution_score": node["contribution_score"],
            }
            for node in global_nodes
        ),
        key=lambda item: item["contribution_score"],
        reverse=True,
    )[:5]

    return {
        "global_nodes": global_nodes,
        "dedup_ratio": round(dedup_ratio, 4),
        "top_contributors": top_contributors,
    }

def build_method_coverage(entries: Iterable[dict[str, object]]) -> tuple[dict[str,
object], str]:
    """Generate method coverage matrix and heatmap recommendations."""

    dimensions = sorted({entry["dimension"] for entry in entries})
    matrix: dict[str, dict[str, dict[str, float]]] = defaultdict(
        lambda: {dim: {"invocations": 0, "tests": 0} for dim in dimensions}
    )

    for entry in entries:
        method = entry["method_id"]
        dim = entry["dimension"]
        matrix[method][dim]["invocations"] += entry["invocations"]
        matrix[method][dim]["tests"] += entry["tests_executed"]

    recommendations: list[dict[str, object]] = []
    for method, dim_data in matrix.items():
        cold_dims: list[str] = []
        for dim, stats in dim_data.items():
            inv = stats["invocations"]
            tests = stats["tests"]
            coverage_ratio = tests / inv if inv else 0.0
            stats["coverage_ratio"] = round(coverage_ratio, 3)
            if inv and coverage_ratio < 0.25 or not inv:
                cold_dims.append(dim)

        if cold_dims:
            recommendations.append(
                {
                    "method_id": method,
                    "cold_dimensions": cold_dims,
                    "action": "Design targeted regression tests for under-covered
dimensions",
                }
            )

    # Build ASCII table
    header = ["Method"] + dimensions
    rows: list[list[str]] = []
    for method in sorted(matrix):
        row = [method]
        for dim in dimensions:
            stats = matrix[method][dim]
            if stats["invocations"]:
                cell = f"{int(stats['invocations'])}/{int(stats['tests'])}"
            else:
                cell = "0/0"
            row.append(cell)
        rows.append(row)

    col_widths = [max(len(row[i]) for row in [header] + rows) for i in range(len(header))]

    def _format_row(row: list[str]) -> str:
        return " | ".join(val.ljust(col_widths[idx]) for idx, val in enumerate(row))

    separator = "-+-".join("-" * width for width in col_widths)
    table_lines = [_format_row(header), separator]
    table_lines.extend(_format_row(row) for row in rows)
    ascii_table = "\n".join(table_lines)

    matrix_serializable = {
        method: {
            dim: {
                "invocations": stats["invocations"],
                "tests": stats["tests"],
                "coverage_ratio": stats.get("coverage_ratio", 0.0),
            }
            for dim, stats in dim_data.items()
        }
        for method, dim_data in matrix.items()
    }

    return (
        {
            "matrix": matrix_serializable,
            "dimensions": dimensions,
            "recommendations": recommendations,
        },
        ascii_table,
    )

SEVERITY_WEIGHTS = {
    "critical": 4,
    "high": 3,
    "medium": 2,
    "low": 1,
}

def analyze_contract_failures(
    entries: Iterable[dict[str, object]], inputs: dict[str, int]
) -> tuple[dict[str, object], list[str]]:
    """Aggregate contract failures into a funnel and narrative."""

    level_stats: dict[str, dict[str, object]] = {}
    method_scores: dict[str, dict[str, object]] = defaultdict(
        lambda: {"severity_score": 0, "total_failures": 0}
    )

    for entry in entries:
        level = entry["level"]
        severity = entry["severity"].lower()
        count = entry["count"]

        stats = level_stats.setdefault(
            level,
            {"by_severity": defaultdict(int), "total_failures": 0},
        )
        stats["by_severity"][severity] += count
        stats["total_failures"] += count

        method_key = f"{entry['method_id']}::{entry['question_id']}"
        method_scores[method_key]["severity_score"] += SEVERITY_WEIGHTS.get(severity, 0) *
 count
        method_scores[method_key]["total_failures"] += count

    for level, stats in level_stats.items():
        entries_prev = inputs.get(level, 0)
        drop_pct = stats["total_failures"] / entries_prev if entries_prev else 0.0
        stats["funnel_drop_pct"] = round(drop_pct * 100, 2)
        stats["by_severity"] = dict(stats["by_severity"])

    top_methods = sorted(
        (
            {
                "method_id": key.split("::")[0],
                "context": key.split("::")[1],
                "severity_score": value["severity_score"],
                "total_failures": value["total_failures"],
            }
            for key, value in method_scores.items()
        ),
        key=lambda item: (item["severity_score"], item["total_failures"]),
        reverse=True,
    )[:5]

    narrative = [
        "Micro layer accumulates the largest absolute failures, primarily from the
assembler and processor modules.",
        "Meso layer exhibits a sharper proportional drop, signalling propagation of
unresolved micro issues into cluster synthesis.",
        "Macro convergence remains fragile with critical severities persisting despite
lower volume.",
        "TeoriaCambio validation spikes as a critical blocker along the causal
verification chain.",
        "Prioritize regression tests around ReportAssembler.generate_meso_cluster to
contain meso escalations.",
    ]

    return {"funnel": level_stats, "top_methods": top_methods}, narrative

@dataclass
class PathStatus:
    path_id: str
    complete: bool
    missing_dimensions: list[str]
    issues: list[str]

def evaluate_causal_paths(data: dict[str, object]) -> dict[str, object]:
    """Verify causal path continuity across dimensions."""

    expected_sequence: list[str] = data["dimension_sequence"]
    complete_paths: list[dict[str, object]] = []
    broken_paths: list[dict[str, object]] = []
    repair_actions: list[str] = []

    for path in data["causal_paths"]:
        dims: list[str] = path["dimensions"]
        missing = [dim for dim in expected_sequence if dim not in dims]
        unexpected = [dim for dim in dims if dim not in expected_sequence]
        issues: list[str] = []

        if dims != expected_sequence:
            # Check for unexpected dimensions (not in expected sequence)
            if unexpected:
                issues.append(
                    "Unexpected dimensions: " + ", ".join(unexpected)
                )

            # Check for length mismatch
            if len(dims) != len(expected_sequence):
                issues.append(
                    f"Length mismatch: expected {len(expected_sequence)} dimensions but
found {len(dims)}"
                )

            # Check for adjacency breaks
            for idx, expected_dim in enumerate(expected_sequence):
                if idx >= len(dims):
                    break
                if dims[idx] != expected_dim:
                    issues.append(
                        f"Expected {expected_dim} at position {idx + 1} but found
{dims[idx]}"
                    )

            if missing:
                issues.append(
                    "Missing dimensions: " + ", ".join(sorted(missing))
                )

        status = PathStatus(
            path_id=path["path_id"],
            complete=not issues and not missing,
            missing_dimensions=missing,
            issues=issues,
        )

        if status.complete:
            complete_paths.append(
                {
                    "path_id": status.path_id,
                    "sequence": path["sequence"],
                }
            )
        else:
            broken_paths.append(
                {
                    "path_id": status.path_id,
                    "missing_dimensions": status.missing_dimensions,
                    "issues": status.issues,
                }
            )
            for dim in status.missing_dimensions:
                repair_actions.append(
                    f"Re-evaluate {dim} in {status.path_id} to restore sequential
continuity"
                )
            for dim in unexpected:
                repair_actions.append(
                    f"Remove unexpected dimension {dim} from {status.path_id}"
                )

    return {
        "complete_paths": complete_paths,
        "broken_paths": broken_paths,
        "repair_actions": sorted(set(repair_actions)),
    }

def run() -> None:
    """Execute all Prompt Cross analyses and print results."""

    data = _load_data()

    evidence = consolidate_evidence(data["qmcm_records"])
    print("=== Prompt Cross – Evidence Registry Consolidation ===")
    print(json.dumps(evidence, indent=2))

    heatmap_json, ascii_table = build_method_coverage(data["method_coverage"])
    print("\n=== Prompt Cross – Method Coverage Heatmap ===")
    print(json.dumps(heatmap_json, indent=2))
    print("\n" + ascii_table)

    funnel_json, narrative = analyze_contract_failures(
        data["contract_failures"], data["funnel_inputs"]
    )
    print("\n=== Prompt Cross – Contract Failure Funnel ===")
    print(json.dumps(funnel_json, indent=2))
    print("\nNarrativa:")
    for line in narrative:
        print(f"- {line}")

    causal = evaluate_causal_paths(data)
    print("\n=== Prompt Cross – Causal Path Integrity ===")
    print(json.dumps(causal, indent=2))

if __name__ == "__main__":
    run()

===== FILE: tools/scan_boundaries.py =====
#!/usr/bin/env python3
"""
AST Scanner for Core Module Boundary Violations

Scans Python modules for:
1. I/O operations (open, json.load/dump, pickle, pandas read_*, etc.)
2. __main__ blocks
3. Side effects on import
4. subprocess, requests, click usage

Usage:
    python tools/scan_boundaries.py --root src --fail-on=io,subprocess,requests,main
                                    --allow-path src/examples src/cli
                                    --sarif out/boundaries.sarif --json
out/violations.json

Exit code 0 if clean, 1 if violations found.
"""

import argparse
import ast
import json
import sys
from datetime import datetime
from pathlib import Path

class BoundaryViolationVisitor(ast.NodeVisitor):
    """AST visitor to detect boundary violations in core modules."""

    # I/O function names to detect
    IO_FUNCTIONS = {
        'open', 'read', 'write',
        'load', 'dump', 'loads', 'dumps',
        'read_csv', 'read_excel', 'read_json', 'read_sql', 'read_parquet',
        'to_csv', 'to_excel', 'to_json', 'to_sql', 'to_parquet',
        'read_text', 'write_text', 'read_bytes', 'write_bytes',
    }

    # Module names that indicate I/O
    IO_MODULES = {
        'pickle', 'json', 'yaml', 'toml', 'csv',
    }

    # Subprocess/network modules
    SUBPROCESS_MODULES = {'subprocess', 'os.system'}
    NETWORK_MODULES = {'requests', 'urllib', 'http', 'httpx'}
    CLI_MODULES = {'click', 'argparse', 'sys.argv'}

    def __init__(self, filename: str) -> None:
        self.filename = filename
        self.violations: list[dict[str, any]] = []
        self.has_main_block = False
        self.io_calls: list[tuple[int, str]] = []
        self.subprocess_calls: list[tuple[int, str]] = []
        self.network_calls: list[tuple[int, str]] = []
        self.cli_usage: list[tuple[int, str]] = []

    def visit_If(self, node: ast.If) -> None:
        """Detect if __name__ == '__main__' blocks."""
        # Check for __name__ == '__main__' pattern
        if isinstance(node.test, ast.Compare) and isinstance(node.test.left, ast.Name) and
 node.test.left.id == '__name__':
            for comparator in node.test.comparators:
                if isinstance(comparator, ast.Constant) and comparator.value ==
'__main__':
                    self.has_main_block = True
                    self.violations.append({
                        'type': 'main_block',
                        'line': node.lineno,
                        'message': f'__main__ block found at line {node.lineno}'
                    })
        self.generic_visit(node)

    def visit_Call(self, node: ast.Call) -> None:
        """Detect I/O function calls."""
        # Direct function calls
        if isinstance(node.func, ast.Name):
            if node.func.id in self.IO_FUNCTIONS:
                self.io_calls.append((node.lineno, node.func.id))
                self.violations.append({
                    'type': 'io_call',
                    'line': node.lineno,
                    'function': node.func.id,
                    'message': f'I/O operation {node.func.id}() at line {node.lineno}'
                })

        # Module.function calls (e.g., json.load)
        elif isinstance(node.func, ast.Attribute) and isinstance(node.func.value,
ast.Name):
            module_name = node.func.value.id
            func_name = node.func.attr

            # Check for I/O
            if module_name in self.IO_MODULES or func_name in self.IO_FUNCTIONS:
                self.io_calls.append((node.lineno, f'{module_name}.{func_name}'))
                self.violations.append({
                    'type': 'io_call',
                    'line': node.lineno,
                    'function': f'{module_name}.{func_name}',
                    'message': f'I/O operation {module_name}.{func_name}() at line
{node.lineno}'
                })
            # Check for subprocess
            if module_name in self.SUBPROCESS_MODULES:
                self.subprocess_calls.append((node.lineno, f'{module_name}.{func_name}'))
                self.violations.append({
                    'type': 'subprocess_call',
                    'line': node.lineno,
                    'function': f'{module_name}.{func_name}',
                    'message': f'Subprocess call {module_name}.{func_name}() at line
{node.lineno}'
                })

            # Check for network
            if module_name in self.NETWORK_MODULES:
                self.network_calls.append((node.lineno, f'{module_name}.{func_name}'))
                self.violations.append({
                    'type': 'network_call',
                    'line': node.lineno,
                    'function': f'{module_name}.{func_name}',
                    'message': f'Network call {module_name}.{func_name}() at line
{node.lineno}'
                })

        self.generic_visit(node)

    def visit_With(self, node: ast.With) -> None:
        """Detect 'with open(...)' patterns."""
        for item in node.items:
            if (isinstance(item.context_expr, ast.Call) and
                isinstance(item.context_expr.func, ast.Name) and
                item.context_expr.func.id == 'open'):
                self.io_calls.append((node.lineno, 'open (with)'))
                self.violations.append({
                    'type': 'io_call',
                    'line': node.lineno,
                    'function': 'open',
                    'message': f'I/O operation: with open(...) at line {node.lineno}'
                })
        self.generic_visit(node)

def scan_file(filepath: Path) -> dict[str, any]:
    """Scan a single Python file for boundary violations."""
    try:
        with open(filepath, encoding='utf-8') as f:
            source = f.read()
    except Exception as e:
        return {
            'file': str(filepath),
            'error': f'Could not read file: {e}',
            'violations': [],
            'clean': False
        }

    try:
        tree = ast.parse(source, filename=str(filepath))
    except SyntaxError as e:
        return {
            'file': str(filepath),
            'error': f'Syntax error: {e}',
            'violations': [],
            'clean': False
        }

    visitor = BoundaryViolationVisitor(str(filepath))
    visitor.visit(tree)

    return {
        'file': str(filepath),
        'violations': visitor.violations,
        'has_main_block': visitor.has_main_block,
        'io_call_count': len(visitor.io_calls),
        'clean': len(visitor.violations) == 0,
        'error': None
    }

def scan_directory(directory: Path, pattern: str = '*.py') -> list[dict[str, any]]:
    """Scan all Python files in a directory."""
    results = []
    for filepath in sorted(directory.rglob(pattern)):
        # Skip __pycache__ and test files
        if '__pycache__' in str(filepath) or 'test_' in filepath.name:
            continue
        results.append(scan_file(filepath))
    return results

def generate_sarif_report(results: list[dict[str, any]], tool_version: str = "1.0.0") ->
dict:
    """Generate SARIF 2.1.0 format report for GitHub annotations."""
    sarif = {
        "version": "2.1.0",
        "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-
spec/master/Schemata/sarif-schema-2.1.0.json",
        "runs": [
            {
                "tool": {
                    "driver": {
                        "name": "BoundaryScanner",
                        "version": tool_version,
                        "informationUri": "https://github.com/kkkkknhh/SAAAAAA",
                        "rules": [
                            {
                                "id": "IO_VIOLATION",
                                "name": "I/O Operation in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not perform I/O operations"
                                },
                                "fullDescription": {
                                    "text": "All I/O operations must be performed through
ports and adapters"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "MAIN_BLOCK",
                                "name": "__main__ Block in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not contain __main__
blocks"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "SUBPROCESS_VIOLATION",
                                "name": "Subprocess Call in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not call subprocess"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "NETWORK_VIOLATION",
                                "name": "Network Call in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not make network calls"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            }
                        ]
                    }
                },
                "results": []
            }
        ]
    }

    for result in results:
        if not result['clean'] and not result.get('error'):
            for violation in result['violations']:
                rule_id = {
                    'io_call': 'IO_VIOLATION',
                    'main_block': 'MAIN_BLOCK',
                    'subprocess_call': 'SUBPROCESS_VIOLATION',
                    'network_call': 'NETWORK_VIOLATION'
                }.get(violation['type'], 'IO_VIOLATION')

                sarif_result = {
                    "ruleId": rule_id,
                    "level": "error",
                    "message": {
                        "text": violation['message']
                    },
                    "locations": [
                        {
                            "physicalLocation": {
                                "artifactLocation": {
                                    "uri": result['file'],
                                    "uriBaseId": "%SRCROOT%"
                                },
                                "region": {
                                    "startLine": violation['line'],
                                    "startColumn": 1
                                }
                            }
                        }
                    ]
                }
                sarif['runs'][0]['results'].append(sarif_result)

    return sarif

def generate_json_report(results: list[dict[str, any]]) -> dict:
    """Generate JSON violations report keyed by file, line, and node type."""
    violations_by_file = {}

    for result in results:
        if not result['clean']:
            file_path = result['file']
            violations_by_file[file_path] = {
                'violations': result['violations'],
                'has_main_block': result.get('has_main_block', False),
                'io_call_count': result.get('io_call_count', 0),
                'error': result.get('error')
            }

    return {
        'timestamp': datetime.now().isoformat(),
        'total_files_scanned': len(results),
        'files_with_violations': len(violations_by_file),
        'total_violations': sum(len(r['violations']) for r in results),
        'violations_by_file': violations_by_file
    }

def should_allow_path(filepath: Path, allowed_paths: list[str]) -> bool:
    """Check if filepath is in any of the allowed paths."""
    filepath_str = str(filepath)
    return any(allowed in filepath_str for allowed in allowed_paths)

def print_report(results: list[dict[str, any]], fail_on_types: set[str] | None = None) ->
int:
    """Print scan results and return exit code."""
    total_files = len(results)
    clean_files = sum(1 for r in results if r['clean'])
    total_violations = sum(len(r['violations']) for r in results)

    if fail_on_types is None:
        fail_on_types = {'io_call', 'main_block', 'subprocess_call', 'network_call'}

    print("=" * 80)
    print("CORE MODULE BOUNDARY SCAN REPORT")
    print("=" * 80)
    print(f"\nFiles scanned: {total_files}")
    print(f"Clean files: {clean_files}")
    print(f"Files with violations: {total_files - clean_files}")
    print(f"Total violations: {total_violations}")
    print()

    if total_violations == 0:
        print("✓ All files are clean! No boundary violations detected.")
        return 0

    print("✗ Violations found:\n")

    # Count violations by type
    violation_counts = {}
    for result in results:
        if not result['clean']:
            for violation in result['violations']:
                vtype = violation['type']
                violation_counts[vtype] = violation_counts.get(vtype, 0) + 1

    print("Violation summary:")
    for vtype, count in sorted(violation_counts.items()):
        marker = "✗" if vtype in fail_on_types else "⚠ "
        print(f"  {marker} {vtype}: {count}")
    print()

    for result in results:
        if not result['clean']:
            print(f"\n{result['file']}")
            if result.get('error'):
                print(f"  ERROR: {result['error']}")
            else:
                for violation in result['violations']:
                    marker = "✗" if violation['type'] in fail_on_types else "⚠ "
                    print(f"  {marker} Line {violation['line']}: {violation['message']}")

    print("\n" + "=" * 80)
    print("REMEDIATION:")
    print("- Move all __main__ blocks to examples/ directory")
    print("- Move all I/O operations to orchestrator/factory.py")
    print("- Core modules should be pure libraries receiving data via contracts")
    print("=" * 80)

    # Determine if we should fail based on fail_on_types
    should_fail = any(
        violation['type'] in fail_on_types
        for result in results
        for violation in result['violations']
    )

    return 1 if should_fail else 0

def main() -> int:
    """Main entry point."""
    parser = argparse.ArgumentParser(
        description='Scan Python modules for boundary violations',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # Basic scan
  python tools/scan_boundaries.py --root src/saaaaaa/core

  # Fail on specific violations only
  python tools/scan_boundaries.py --root src --fail-on io,main

  # Allow specific paths
  python tools/scan_boundaries.py --root src --fail-on io,subprocess,requests,main \\
                                  --allow-path src/examples src/cli

  # Generate SARIF and JSON reports
  python tools/scan_boundaries.py --root src/saaaaaa/core \\
                                  --sarif out/boundaries.sarif \\
                                  --json out/violations.json
        """
    )

    parser.add_argument(
        '--root',
        type=str,
        required=True,
        help='Root directory to scan'
    )

    parser.add_argument(
        '--fail-on',
        type=str,
        default='io,main,subprocess,network',
        help='Comma-separated list of violation types to fail on (io, main, subprocess,
network)'
    )

    parser.add_argument(
        '--allow-path',
        nargs='+',
        default=[],
        help='Paths to exclude from scanning (e.g., src/examples src/cli)'
    )

    parser.add_argument(
        '--sarif',
        type=str,
        help='Output SARIF report to this file'
    )

    parser.add_argument(
        '--json',
        type=str,
        help='Output JSON violations report to this file'
    )

    # Legacy positional argument support
    if len(sys.argv) == 2 and not sys.argv[1].startswith('--'):
        # Old style: python scan_boundaries.py <directory>
        target_path = Path(sys.argv[1])
        fail_on_types = {'io_call', 'main_block', 'subprocess_call', 'network_call'}
        allowed_paths = []
        sarif_output = None
        json_output = None
    else:
        args = parser.parse_args()
        target_path = Path(args.root)
        fail_on_types = set()
        for vtype in args.fail_on.split(','):
            vtype = vtype.strip()
            if vtype == 'io':
                fail_on_types.add('io_call')
            elif vtype == 'main':
                fail_on_types.add('main_block')
            elif vtype == 'subprocess':
                fail_on_types.add('subprocess_call')
            elif vtype == 'network':
                fail_on_types.add('network_call')

        allowed_paths = args.allow_path
        sarif_output = args.sarif
        json_output = args.json

    if not target_path.exists():
        print(f"Error: Directory {target_path} does not exist")
        return 1

    results = scan_directory(target_path)

    # Filter results based on allowed paths
    if allowed_paths:
        filtered_results = []
        for result in results:
            if not should_allow_path(Path(result['file']), allowed_paths):
                filtered_results.append(result)
            else:
                # Mark as clean if in allowed path
                result['clean'] = True
                result['violations'] = []
                filtered_results.append(result)
        results = filtered_results

    # Generate SARIF report if requested
    if sarif_output:
        sarif_path = Path(sarif_output)
        sarif_path.parent.mkdir(parents=True, exist_ok=True)
        sarif_data = generate_sarif_report(results)
        with open(sarif_path, 'w', encoding='utf-8') as f:
            json.dump(sarif_data, f, indent=2)
        print(f"SARIF report written to {sarif_output}")

    # Generate JSON report if requested
    if json_output:
        json_path = Path(json_output)
        json_path.parent.mkdir(parents=True, exist_ok=True)
        json_data = generate_json_report(results)
        with open(json_path, 'w', encoding='utf-8') as f:
            json.dump(json_data, f, indent=2)
        print(f"JSON report written to {json_output}")

    return print_report(results, fail_on_types)

if __name__ == '__main__':
    sys.exit(main())

===== FILE: tools/scan_core_purity.py =====
```

```python
#!/usr/bin/env python3
"""
Core Purity Scanner - Ensures core modules follow functional purity principles.

Checks:
1. No I/O operations in core modules (print, open, file operations)
2. No __main__ blocks in core modules
3. No direct database or network calls
"""

import ast
import sys
from pathlib import Path
from typing import List, Tuple

try:
    from saaaaaa.config.paths import PROJECT_ROOT
except Exception:  # pragma: no cover - bootstrap fallback
    PROJECT_ROOT = Path(__file__).resolve().parents[1]

# Directories that must maintain purity
CORE_PATHS = [
    "src/saaaaaa/core",
]

# Forbidden operations (allowing open for config loading, but not print/input)
FORBIDDEN_FUNCTIONS = {
    "print", "input",
}

FORBIDDEN_IMPORTS = {
    "requests", "urllib", "socket", "sqlalchemy",
}


class PurityChecker(ast.NodeVisitor):
    """AST visitor to detect impure operations."""

    def __init__(self, filepath: Path):
        self.filepath = filepath
        self.violations: List[Tuple[int, str]] = []

    def visit_Call(self, node: ast.Call):
        """Check for forbidden function calls."""
        if isinstance(node.func, ast.Name):
            if node.func.id in FORBIDDEN_FUNCTIONS:
                self.violations.append(
                    (node.lineno, f"Forbidden function: {node.func.id}")
                )
        self.generic_visit(node)

    def visit_If(self, node: ast.If):
        """Check for __main__ blocks."""
        if isinstance(node.test, ast.Compare):
            if isinstance(node.test.left, ast.Name):
                if node.test.left.id == "__name__":
                    self.violations.append(
                        (node.lineno, "Forbidden __main__ block in core module")
                    )
        self.generic_visit(node)

    def visit_Import(self, node: ast.Import):
        """Check for forbidden imports."""
        for alias in node.names:
            if any(forbidden in alias.name for forbidden in FORBIDDEN_IMPORTS):
                self.violations.append(
                    (node.lineno, f"Forbidden import: {alias.name}")
                )
        self.generic_visit(node)

    def visit_ImportFrom(self, node: ast.ImportFrom):
        """Check for forbidden imports."""
        if node.module:
            if any(forbidden in node.module for forbidden in FORBIDDEN_IMPORTS):
                self.violations.append(
                    (node.lineno, f"Forbidden import from: {node.module}")
                )
        self.generic_visit(node)


def check_file_purity(filepath: Path) -> List[Tuple[int, str]]:
    """Check a single file for purity violations."""
    try:
        with open(filepath, "r", encoding="utf-8") as f:
            tree = ast.parse(f.read(), filename=str(filepath))

        checker = PurityChecker(filepath)
        checker.visit(tree)
        return checker.violations
    except SyntaxError as e:
        return [(e.lineno or 0, f"Syntax error: {e.msg}")]
    except Exception as e:
        return [(0, f"Error parsing file: {e}")]


def main() -> int:
    """Scan all core modules for purity violations."""
    repo_root = PROJECT_ROOT
    violations_found = False

    for core_path_str in CORE_PATHS:
        core_path = repo_root / core_path_str

        if not core_path.exists():
            print(f"⚠  Core path not found: {core_path}")
            continue

        print(f"Scanning {core_path_str}...")

        for py_file in core_path.rglob("*.py"):
            violations = check_file_purity(py_file)

            if violations:
                violations_found = True
                rel_path = py_file.relative_to(repo_root)
                print(f"\n✗ {rel_path}")
                for lineno, msg in violations:
                    print(f"  Line {lineno}: {msg}")

    if violations_found:
        print("\n✗ Core purity violations detected")
        return 1
    else:
        print("✓ All core modules are pure")
        return 0


if __name__ == "__main__":
    sys.exit(main())

===== FILE: tools/testing/__init__.py =====
"""Testing utilities."""

===== FILE: tools/testing/boot_check.py =====
#!/usr/bin/env python3
"""
```

```
Boot check script for CI and pre-production environments.

Validates that all modules load correctly, runtime validators initialize,
and the registry is complete without ClassNotFoundError.

Usage:
    python tools/testing/boot_check.py
    python tools/testing/boot_check.py --verbose
"""

import importlib
import sys
import traceback
from pathlib import Path

# Add project root to Python path

# Modules to validate
CORE_MODULES = [
    "orchestrator",
    "scoring",
    "recommendation_engine",
    "validation_engine",
    "policy_processor",
    "embedding_policy",
    "semantic_chunking_policy",
]

OPTIONAL_MODULES = [
    "derek_beach",
    "contradiction_deteccion",
    "teoria_cambio",
    "financiero_viabilidad_tablas",
    "macro_prompts",
    "micro_prompts",
]

def check_module_import(module_name: str, verbose: bool = False) -> tuple[bool, str]:
    """
    Try to import a module and return success status.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print(f"  Importing {module_name}...", end=" ")

        importlib.import_module(module_name)

        if verbose:
            print("✓")

        return True, ""
    except ModuleNotFoundError as e:
        error = f"Module not found: {e}"
        if verbose:
            print(f"✗ {error}")
        return False, error
    except ImportError as e:
        error = f"Import error: {e}"
        if verbose:
            print(f"✗ {error}")
        return False, error
    except Exception as e:
        error = f"Unexpected error: {e}"
        if verbose:
            print(f"✗ {error}")
        if verbose:
            traceback.print_exc()
        return False, error

def check_registry_validation(verbose: bool = False) -> tuple[bool, str]:
    """
    Validate that the orchestrator registry loads without ClassNotFoundError.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print("  Validating orchestrator registry...", end=" ")

        # Try to import and access the registry
        from saaaaaa.core.orchestrator import registry

        # Try to validate all classes (if method exists)
        if hasattr(registry, 'validate_all_classes'):
            registry.validate_all_classes()

        if verbose:
            print("✓")

        return True, ""
    except NameError as e:
        if "ClassNotFoundError" in str(e) or "not defined" in str(e):
            error = f"ClassNotFoundError in registry: {e}"
            if verbose:
                print(f"✗ {error}")
            return False, error
        raise
    except AttributeError as e:
        # Module or registry doesn't exist - return as informational
        if verbose:
            print(f"⚠ Registry validation not available: {e}")
        return True, ""  # Don't fail if registry validation not implemented
    except Exception as e:
        error = f"Registry validation error: {e}"
        if verbose:
            print(f"✗ {error}")
            traceback.print_exc()
        return False, error

def check_runtime_validators(verbose: bool = False) -> tuple[bool, str]:
    """
    Validate that runtime validators initialize correctly.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print("  Initializing runtime validators...", end=" ")

        # Try to import and initialize validators
        from saaaaaa.validation.validation_engine import RuntimeValidator

        validator = RuntimeValidator()

        # Try to run health check if available
        if hasattr(validator, 'health_check'):
            validator.health_check()

        if verbose:
            print("✓")
        return True, ""
    except ImportError:
        # validation_engine doesn't exist or RuntimeValidator not available
        if verbose:
            print("⚠ Runtime validators not available (skipping)")
        return True, ""  # Don't fail if not implemented
    except Exception as e:
        error = f"Runtime validator initialization error: {e}"
        if verbose:
            print(f"✗ {error}")
            traceback.print_exc()
        return False, error

def run_boot_checks(verbose: bool = False) -> int:
    """
    Run all boot checks.

    Returns:
        Exit code (0 = success, 1 = failure)
    """
    print("=" * 60)
    print("Boot Check - Module and Runtime Validation")
    print("=" * 60)

    all_passed = True
    failed_checks = []

    # Check core modules
    print("\nChecking core modules:")
    core_failed = []
    for module in CORE_MODULES:
        success, error = check_module_import(module, verbose)
        if not success:
            all_passed = False
            core_failed.append(f"{module}: {error}")
            failed_checks.append(f"Core module {module} failed to load")

    if not verbose:
        if core_failed:
            print(f"  ✗ {len(core_failed)} core module(s) failed to load")
            for failure in core_failed[:3]:
                print(f"    - {failure}")
            if len(core_failed) > 3:
                print(f"    ... and {len(core_failed) - 3} more")
        else:
            print(f"  ✓ All {len(CORE_MODULES)} core modules loaded successfully")

    # Check optional modules
    print("\nChecking optional modules:")
    optional_failed = []
    for module in OPTIONAL_MODULES:
        success, error = check_module_import(module, verbose)
        if not success:
            optional_failed.append(f"{module}: {error}")
            # Don't fail overall for optional modules

    if not verbose:
        loaded_count = len(OPTIONAL_MODULES) - len(optional_failed)
        print(f"  ✓ {loaded_count}/{len(OPTIONAL_MODULES)} optional modules loaded")
        if optional_failed:
            print(f"  ⚠ {len(optional_failed)} optional module(s) not available")

    # Check registry
    print("\nChecking orchestrator registry:")
    success, error = check_registry_validation(verbose)
    if not success:
        all_passed = False
        failed_checks.append(f"Registry validation failed: {error}")
    elif not verbose:
        print("  ✓ Registry validation passed")

    # Check runtime validators
    print("\nChecking runtime validators:")
    success, error = check_runtime_validators(verbose)
    if not success:
        all_passed = False
        failed_checks.append(f"Runtime validator initialization failed: {error}")
    elif not verbose:
        print("  ✓ Runtime validators initialized successfully")

    # Summary
    print("\n" + "=" * 60)
    if all_passed:
        print("✓ All boot checks PASSED")
        print("=" * 60)
        return 0
    else:
        print("✗ Some boot checks FAILED")
        print("\nFailed checks:")
        for check in failed_checks:
            print(f"  - {check}")
        print("=" * 60)
        return 1

def main() -> None:
    verbose = "--verbose" in sys.argv or "-v" in sys.argv

    exit_code = run_boot_checks(verbose)
    sys.exit(exit_code)

if __name__ == "__main__":
    main()

===== FILE: tools/validation/__init__.py =====
"""Validation tools."""

===== FILE: tools/validation/validate_build_hygiene.py =====
#!/usr/bin/env python3
"""
Validation script for build hygiene checklist.
Verifies that the repository follows all build hygiene requirements.
"""

import re
import sys
from pathlib import Path

def check_python_version_pin() -> bool:
    """Check that Python version is pinned to 3.12.x."""
    print("✓ Checking Python version pin...")

    # Check .python-version
    python_version_file = Path(".python-version")
    if not python_version_file.exists():
        print("  ✗ Missing .python-version file")
        return False

    version = python_version_file.read_text().strip()
    if not version.startswith("3.12"):
        print(f"  ✗ .python-version should be 3.12.x, got {version}")
        return False

    # Check pyproject.toml
    pyproject = Path("pyproject.toml").read_text()
    if 'requires-python = "~=3.12.0"' not in pyproject:
        print("  ✗ pyproject.toml should have requires-python = \"~=3.12.0\"")
        return False

    if 'pythonVersion = "3.12"' not in pyproject:
        print("  ✗ pyproject.toml should have pythonVersion = \"3.12\"")
        return False

    print("  ✓ Python version properly pinned to 3.12.x")
    return True

def check_pinned_dependencies() -> bool:
    """Check that all dependencies are pinned to exact versions."""
    print("✓ Checking dependency pinning...")

    requirements = Path("requirements.txt")
    if not requirements.exists():
        print("  ✗ Missing requirements.txt")
        return False

    constraints = Path("constraints.txt")
    if not constraints.exists():
        print("  ✗ Missing constraints.txt")
        return False

    # Check for wildcards or open ranges in requirements.txt
    content = requirements.read_text()
    lines = [line.strip() for line in content.split('\n')
            if line.strip() and not line.strip().startswith('#')]

    bad_patterns = []
    for line in lines:
        # Check for wildcard or open ranges in version specifiers
        # Look for these patterns after package name and optional extras
        # Match patterns like: package>=1.0, package~=1.0, package>1.0, package<2.0,
package*
        if (re.search(r'(>=|~=|>|<|\*)', line) and
            re.search(r'[^\[]*(\[.*\])?\s*(>=|~=|>|<|\*)', line)):
            # Further validate it's in version spec position, not in package name
            # Package name format: name[extras]==version
            bad_patterns.append(line)

    if bad_patterns:
        print("  ✗ Found wildcards or open ranges in requirements.txt:")
        for pattern in bad_patterns:
            print(f"    - {pattern}")
        return False

    print(f"  ✓ All {len(lines)} dependencies properly pinned with exact versions")
    print("  ✓ constraints.txt exists")
    return True

def check_directory_structure() -> bool:
    """Check that required directories exist with __init__.py files."""
    print("✓ Checking directory structure...")

    required_dirs = [
        "orchestrator",
        "executors",
        "contracts",
        "tests",
        "tools",
        "examples",
        "src/saaaaaa/core",
    ]

    missing_dirs = []
    missing_init = []

    for dir_path in required_dirs:
        path = Path(dir_path)
        if not path.exists():
            missing_dirs.append(dir_path)
        elif not (path / "__init__.py").exists():
            missing_init.append(dir_path)

    if missing_dirs:
        print("  ✗ Missing directories:")
        for d in missing_dirs:
            print(f"    - {d}")
        return False

    if missing_init:
        print("  ✗ Missing __init__.py in:")
        for d in missing_init:
            print(f"    - {d}")
        return False

    print("  ✓ All required directories exist with __init__.py files")
    return True

def check_pythonpath_config() -> bool:
    """Check that setup.py exists for proper PYTHONPATH configuration."""
    print("✓ Checking PYTHONPATH configuration...")

    setup_py = Path("setup.py")
    if not setup_py.exists():
        print("  ✗ Missing setup.py for pip install -e .")
        return False

    content = setup_py.read_text()
    if 'find_packages' not in content:
        print("  ✗ setup.py should use find_packages()")
        return False

    print("  ✓ setup.py exists for editable installation")
    return True

def check_centralized_config() -> bool:
    """Check that centralized configuration exists."""
    print("✓ Checking centralized configuration...")

    settings = Path("orchestrator/settings.py")
    if not settings.exists():
        print("  ✗ Missing orchestrator/settings.py")
        return False

    env_example = Path(".env.example")
    if not env_example.exists():
        print("  ✗ Missing .env.example")
        return False

    gitignore = Path(".gitignore")
    if gitignore.exists():
        content = gitignore.read_text()
        if ".env" not in content:
            print("  ✗ .gitignore should exclude .env files")
            return False

    print("  ✓ Centralized config exists (orchestrator/settings.py, .env.example)")
    print("  ✓ .env properly excluded in .gitignore")
    return True

def main() -> int:
    """Run all validation checks."""
    print("=" * 60)
    print("Build Hygiene Validation")
    print("=" * 60)
    print()

    checks = [
        check_python_version_pin,
        check_pinned_dependencies,
        check_directory_structure,
        check_pythonpath_config,
        check_centralized_config,
    ]

    results = []
    for check in checks:
        try:
            results.append(check())
            print()
        except Exception as e:
            print(f"  ✗ Error running check: {e}")
            results.append(False)
            print()

    print("=" * 60)
    if all(results):
        print("✓ All build hygiene checks passed!")
        print("=" * 60)
        return 0
    else:
        failed = sum(1 for r in results if not r)
        print(f"✗ {failed} check(s) failed")
        print("=" * 60)
        return 1

if __name__ == "__main__":
    sys.exit(main())

===== FILE: tools/validation/validate_scoring_parity.py =====
#!/usr/bin/env python3
"""
Validate scoring parity across modalities.

This script ensures that:
1. Normalization formulas are correct for each modality
2. Quality thresholds are identical across all modalities
3. Boundary conditions produce correct quality levels
4. No modality has an unfair advantage at quality boundaries

Usage:
    python tools/validation/validate_scoring_parity.py
    python tools/validation/validate_scoring_parity.py --verbose
"""

import sys

# Quality thresholds (must be identical across all modalities)
QUALITY_THRESHOLDS = {
    "EXCELENTE": 0.85,
    "BUENO": 0.70,
    "ACEPTABLE": 0.55,
    "INSUFICIENTE": 0.00
}

# Modality score ranges
MODALITY_RANGES = {
    "TYPE_A": (0, 4),
    "TYPE_B": (0, 3),
    "TYPE_C": (0, 3),
    "TYPE_D": (0, 3),
    "TYPE_E": (0, 3),
    "TYPE_F": (0, 3),
}

def normalize_score(raw_score: float, modality: str) -> float:
    """Normalize a raw score to [0, 1] range based on modality."""
    min_score, max_score = MODALITY_RANGES[modality]
    if raw_score < min_score or raw_score > max_score:
        raise ValueError(f"Score {raw_score} out of range for {modality}: [{min_score},
{max_score}]")
    return (raw_score - min_score) / (max_score - min_score)

def determine_quality_level(normalized_score: float) -> str:
    """Determine quality level from normalized score."""
    # Use small epsilon for floating point comparison
    epsilon = 1e-9
    if normalized_score >= QUALITY_THRESHOLDS["EXCELENTE"] - epsilon:
        return "EXCELENTE"
    elif normalized_score >= QUALITY_THRESHOLDS["BUENO"] - epsilon:
        return "BUENO"
    elif normalized_score >= QUALITY_THRESHOLDS["ACEPTABLE"] - epsilon:
        return "ACEPTABLE"
    else:
        return "INSUFICIENTE"

def test_normalization_formulas() -> bool:
    """Test that normalization formulas are correct."""
    print("Testing normalization formulas...")

    test_cases = [
        ("TYPE_A", 0, 0.0),
        ("TYPE_A", 2, 0.5),
        ("TYPE_A", 4, 1.0),
        ("TYPE_B", 0, 0.0),
        ("TYPE_B", 1.5, 0.5),
        ("TYPE_B", 3, 1.0),
        ("TYPE_C", 0, 0.0),
        ("TYPE_C", 1.5, 0.5),
        ("TYPE_C", 3, 1.0),
    ]

    passed = 0
    failed = 0

    for modality, raw, expected in test_cases:
        actual = normalize_score(raw, modality)
        if abs(actual - expected) < 0.001:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  ✓ {modality}: {raw} → {actual:.3f} (expected {expected:.3f})")
        else:
            failed += 1
            print(f"  ✗ {modality}: {raw} → {actual:.3f} (expected {expected:.3f})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_parity_at_thresholds() -> bool:
    """Test that all modalities produce the same quality level at threshold scores."""
    print("\nTesting parity at quality thresholds...")

    # Calculate equivalent raw scores for each threshold
    test_cases = []
    for quality, threshold in QUALITY_THRESHOLDS.items():
        if quality == "INSUFICIENTE":
            continue  # Skip lower bound

        for modality, (min_score, max_score) in MODALITY_RANGES.items():
            raw_score = min_score + threshold * (max_score - min_score)
            test_cases.append((modality, raw_score, quality))
    passed = 0
    failed = 0

    for modality, raw_score, expected_quality in test_cases:
        normalized = normalize_score(raw_score, modality)
        actual_quality = determine_quality_level(normalized)

        if actual_quality == expected_quality:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  ✓ {modality} at {raw_score:.2f} → {actual_quality}")
        else:
            failed += 1
            print(f"  ✗ {modality} at {raw_score:.2f} → {actual_quality} (expected
{expected_quality})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_boundary_conditions() -> bool:
    """Test boundary conditions (just above/below thresholds)."""
    print("\nTesting boundary conditions...")

    # Test scores just below and just above EXCELENTE threshold

    test_cases = [
        # Just below EXCELENTE (should be BUENO)
        ("TYPE_A", 3.396, "BUENO"),  # 3.396/4 = 0.849
        ("TYPE_B", 2.547, "BUENO"),  # 2.547/3 = 0.849

        # Just at EXCELENTE threshold
        ("TYPE_A", 3.4, "EXCELENTE"),  # 3.4/4 = 0.85
        ("TYPE_B", 2.55, "EXCELENTE"),  # 2.55/3 = 0.85

        # Just above EXCELENTE
        ("TYPE_A", 3.404, "EXCELENTE"),  # 3.404/4 = 0.851
        ("TYPE_B", 2.553, "EXCELENTE"),  # 2.553/3 = 0.851
    ]

    passed = 0
    failed = 0

    for modality, raw_score, expected_quality in test_cases:
        normalized = normalize_score(raw_score, modality)
        actual_quality = determine_quality_level(normalized)

        if actual_quality == expected_quality:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  ✓ {modality}: {raw_score:.3f} (norm={normalized:.4f}) →
{actual_quality}")
        else:
            failed += 1
            print(f"  ✗ {modality}: {raw_score:.3f} (norm={normalized:.4f}) →
{actual_quality} (expected {expected_quality})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_no_unfair_advantage() -> bool:
    """Test that no modality has an unfair advantage at boundaries."""
    print("\nTesting for unfair advantages...")

    # For each quality threshold, calculate the "difficulty" (raw score needed)
    # relative to the maximum possible score
    difficulties = {}
    for quality, threshold in QUALITY_THRESHOLDS.items():
        if quality == "INSUFICIENTE":
            continue

        difficulties[quality] = {}
        for modality, (min_score, max_score) in MODALITY_RANGES.items():
            raw_needed = min_score + threshold * (max_score - min_score)
            relative_difficulty = raw_needed / max_score
            difficulties[quality][modality] = relative_difficulty

    passed = 0
    failed = 0

    for quality, modality_difficulties in difficulties.items():
        # All modalities should have the same relative difficulty
        values = list(modality_difficulties.values())
        max_diff = max(values) - min(values)

        if max_diff < 0.001:  # Allow 0.1% variance
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  ✓ {quality}: all modalities have equal difficulty (max diff:
{max_diff:.6f})")
        else:
            failed += 1
            print(f"  ✗ {quality}: modalities have unequal difficulty (max diff:
{max_diff:.6f})")
            for modality, diff in modality_difficulties.items():
                print(f"      {modality}: {diff:.6f}")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def main() -> int:
    """Run all parity validation tests."""
    print("=" * 60)
    print("Scoring Parity Validation")
    print("=" * 60)

    all_passed = True

    # Run all tests
    all_passed &= test_normalization_formulas()
    all_passed &= test_parity_at_thresholds()
    all_passed &= test_boundary_conditions()
    all_passed &= test_no_unfair_advantage()

    print("\n" + "=" * 60)
    if all_passed:
        print("✓ All parity validation tests PASSED")
        print("=" * 60)
        return 0
    else:
        print("✗ Some parity validation tests FAILED")
        print("=" * 60)
        return 1

if __name__ == "__main__":
    sys.exit(main())
```