```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 28
================================================================================
Generated: 2025-12-07T06:17:36.486739
Files in this batch: 17
================================================================================


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

"""
Processing Layer Test Suite

Property-based tests using Hypothesis for semantic chunking determinism,
SPC ingestion pipeline idempotency, embedding policy consistency, and converter invariants.
Plus integration tests for quality gates, policy processor end-to-end execution,
and aggregation pipeline correctness.

Test Files:
- test_semantic_chunking_properties.py: Semantic chunking determinism
- test_spc_ingestion_properties.py: SPC ingestion idempotency
- test_embedding_policy_properties.py: Embedding consistency
- test_converter_properties.py: Converter invariants (to be created)
- test_quality_gates_integration.py: Quality gates integration (to be created)
- test_policy_processor_integration.py: End-to-end policy processing (to be created)
- test_aggregation_integration.py: Aggregation pipeline (to be created)
"""



================================================================================
FILE: tests/processing/test_embedding_policy_properties.py
================================================================================

"""
Property-Based Tests for Embedding Policy Consistency

Verifies that embedding generation is consistent, deterministic, and maintains
mathematical properties expected from embeddings.
"""

import numpy as np
import pytest
from hypothesis import HealthCheck, assume, given, settings
from hypothesis import strategies as st
from scipy.spatial.distance import cosine

from src.farfan_pipeline.processing.embedding_policy import (
    AdvancedSemanticChunker,
    BayesianNumericalAnalyzer,
    ChunkingConfig,
)


```

```
@st.composite
def valid_text(draw):
    """Generate valid text for embedding."""
    return draw(st.text(
        alphabet=st.characters(whitelist_categories=('L', 'N', 'P', 'Z')),
        min_size=20,
        max_size=500
    ))


@st.composite
def chunking_config(draw):
    """Generate valid ChunkingConfig instances."""
    chunk_size = draw(st.integers(min_value=128, max_value=1024))
    return ChunkingConfig(
        chunk_size=chunk_size,
        chunk_overlap=draw(st.integers(min_value=16, max_value=chunk_size // 2)),
        min_chunk_size=draw(st.integers(min_value=32, max_value=chunk_size // 4)),
        respect_boundaries=draw(st.booleans()),
        preserve_tables=draw(st.booleans()),
        detect_lists=draw(st.booleans()),
        section_aware=draw(st.booleans())
    )


class TestEmbeddingConsistency:
    """Property-based tests for embedding consistency."""

    @given(text=valid_text())
    @settings(max_examples=20, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
    def test_embedding_dimensions_are_consistent(self, text):
        """Property: All embeddings have consistent dimensions."""
        assume(len(text) > 50)

        config = ChunkingConfig()
        chunker = AdvancedSemanticChunker(config)

        chunks = chunker.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        if len(chunks) > 0 and len(chunks[0].get("embedding", [])) > 0:
            expected_dim = len(chunks[0]["embedding"])

            for chunk in chunks:
                if len(chunk.get("embedding", [])) > 0:
                    assert len(chunk["embedding"]) == expected_dim, \
                        "All embeddings must have same dimension"

    @given(
        text=valid_text(),
        seed=st.integers(min_value=0, max_value=1000)
    )
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_embeddings_are_deterministic(self, text, seed):
        """Property: Same text produces same embeddings."""
        assume(len(text) > 50)

        np.random.seed(seed)
        config = ChunkingConfig()
        chunker1 = AdvancedSemanticChunker(config)
        chunks1 = chunker1.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        np.random.seed(seed)
        chunker2 = AdvancedSemanticChunker(config)
        chunks2 = chunker2.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        assert len(chunks1) == len(chunks2)

        for c1, c2 in zip(chunks1, chunks2, strict=False):
            emb1 = c1.get("embedding", [])
            emb2 = c2.get("embedding", [])

            if len(emb1) > 0 and len(emb2) > 0:
                np.testing.assert_array_almost_equal(emb1, emb2, decimal=4)

    @given(text=valid_text())
    @settings(max_examples=15, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
    def test_embeddings_are_normalized(self, text):
        """Property: Embeddings should be unit vectors or close to it."""
        assume(len(text) > 50)

        config = ChunkingConfig()
        chunker = AdvancedSemanticChunker(config)

        chunks = chunker.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        for chunk in chunks:
            emb = chunk.get("embedding", [])
            if len(emb) > 0:
                norm = np.linalg.norm(emb)
                assert 0.8 <= norm <= 1.2, \
                    f"Embedding norm should be close to 1: {norm}"

    @given(
        text1=valid_text(),
        text2=valid_text()
    )
    @settings(max_examples=10, deadline=6000, suppress_health_check=[HealthCheck.too_slow])
    def test_similar_texts_have_similar_embeddings(self, text1, text2):
        """Property: Similar texts have similar embeddings."""
        assume(len(text1) > 50 and len(text2) > 50)

        words1 = set(text1.lower().split())
        words2 = set(text2.lower().split())

        if len(words1) > 0 and len(words2) > 0:
            overlap = len(words1 & words2) / max(len(words1), len(words2))
            assume(overlap > 0.5)

            config = ChunkingConfig()
            chunker = AdvancedSemanticChunker(config)

            chunks1 = chunker.chunk_document(
                text=text1,
                document_metadata={"doc_id": "doc1"}
            )
            chunks2 = chunker.chunk_document(
                text=text2,
                document_metadata={"doc_id": "doc2"}
            )

            if len(chunks1) > 0 and len(chunks2) > 0:
                emb1 = chunks1[0].get("embedding", [])
                emb2 = chunks2[0].get("embedding", [])

                if len(emb1) > 0 and len(emb2) > 0:
                    similarity = 1 - cosine(emb1, emb2)
                    assert similarity > 0.2, \
                        f"Similar texts should have similar embeddings: {similarity}"


class TestChunkingConfigurationInvariants:
    """Invariant tests for chunking configuration."""

    @given(config=chunking_config())
    @settings(max_examples=30, deadline=1000)
    def test_config_invariants_hold(self, config):
        """Property: Configuration parameters maintain valid relationships."""
        assert config.chunk_size > 0, "Chunk size must be positive"
        assert config.chunk_overlap >= 0, "Overlap must be non-negative"
        assert config.chunk_overlap < config.chunk_size, \
            "Overlap must be less than chunk size"
        assert config.min_chunk_size > 0, "Min chunk size must be positive"
        assert config.min_chunk_size <= config.chunk_size, \
            "Min chunk size must not exceed max chunk size"

    @given(
        text=valid_text(),
        config=chunking_config()
    )
    @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunker_respects_config(self, text, config):
        """Property: Chunker respects configuration parameters."""
        assume(len(text) > config.chunk_size)

        chunker = AdvancedSemanticChunker(config)

        chunks = chunker.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        for chunk in chunks:
            token_count = chunk.get("token_count", 0)

            if config.min_chunk_size:
                assert token_count >= config.min_chunk_size * 0.5 or len(chunks) == 1, \
                    f"Chunk smaller than min size: {token_count} < {config.min_chunk_size}"


class TestBayesianNumericalAnalyzerProperties:
    """Property-based tests for Bayesian numerical analysis."""

    @given(
        values=st.lists(st.floats(min_value=0.0, max_value=1.0), min_size=1, max_size=50),
        prior_alpha=st.floats(min_value=0.1, max_value=10.0),
        prior_beta=st.floats(min_value=0.1, max_value=10.0)
    )
    @settings(max_examples=50, deadline=1000)
    def test_bayesian_evaluation_produces_valid_credible_intervals(self, values, prior_alpha, prior_beta):
        """Property: Credible intervals are valid."""
        analyzer = BayesianNumericalAnalyzer(prior_alpha=prior_alpha, prior_beta=prior_beta)

        result = analyzer.analyze_proportion(
            successes=sum(values),
            trials=len(values),
            context="test"
        )

        lower, upper = result["credible_interval_95"]

        assert 0.0 <= lower <= 1.0, f"Lower bound out of range: {lower}"
        assert 0.0 <= upper <= 1.0, f"Upper bound out of range: {upper}"
        assert lower <= upper, f"Lower bound exceeds upper bound: {lower} > {upper}"
        assert lower <= result["point_estimate"] <= upper, \
            "Point estimate must be within credible interval"

    @given(
        successes=st.integers(min_value=0, max_value=100),
        trials=st.integers(min_value=1, max_value=100)
    )
    @settings(max_examples=50, deadline=1000)
    def test_bayesian_evaluation_is_bounded(self, successes, trials):
        """Property: Bayesian evaluation produces bounded results."""
        assume(successes <= trials)

        analyzer = BayesianNumericalAnalyzer()

        result = analyzer.analyze_proportion(
            successes=successes,
            trials=trials,
            context="test"
        )

        assert 0.0 <= result["point_estimate"] <= 1.0
        assert result["evidence_strength"] in ["weak", "moderate", "strong", "very_strong"]
        assert 0.0 <= result["numerical_coherence"] <= 1.0

    @given(
        successes=st.integers(min_value=0, max_value=100),
        trials=st.integers(min_value=1, max_value=100)
    )
    @settings(max_examples=30, deadline=1000)
    def test_more_data_reduces_uncertainty(self, successes, trials):
        """Property: More data should reduce uncertainty (narrower intervals)."""
        assume(successes <= trials and trials >= 10)

        analyzer = BayesianNumericalAnalyzer()

        result_few = analyzer.analyze_proportion(
            successes=successes // 2,
            trials=trials // 2,
            context="test"
        )

        result_many = analyzer.analyze_proportion(
            successes=successes,
            trials=trials,
            context="test"
        )

        width_few = result_few["credible_interval_95"][1] - result_few["credible_interval_95"][0]
        width_many = result_many["credible_interval_95"][1] - result_many["credible_interval_95"][0]

        assert width_many <= width_few * 1.5, \
            "More data should not significantly increase uncertainty"


class TestPDQContextInference:
    """Property-based tests for PDQ context inference."""

    @given(text=valid_text())
    @settings(max_examples=15, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
    def test_pdq_context_structure_is_valid(self, text):
        """Property: PDQ context has valid structure when inferred."""
        assume(len(text) > 50)

        config = ChunkingConfig()
        chunker = AdvancedSemanticChunker(config)

        chunks = chunker.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        for chunk in chunks:
            pdq = chunk.get("pdq_context")
            if pdq is not None:
                assert "question_unique_id" in pdq
                assert "policy" in pdq
                assert "dimension" in pdq
                assert "question" in pdq
                assert "rubric_key" in pdq

                assert pdq["policy"].startswith("PA")
                assert pdq["dimension"].startswith("DIM")
                assert isinstance(pdq["question"], int)

    @given(text=valid_text())
    @settings(max_examples=10, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
    def test_pdq_inference_is_stable(self, text):
        """Property: PDQ inference is stable across runs."""
        assume(len(text) > 50)

        config = ChunkingConfig()
        chunker1 = AdvancedSemanticChunker(config)
        chunker2 = AdvancedSemanticChunker(config)

        chunks1 = chunker1.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )
        chunks2 = chunker2.chunk_document(
            text=text,
            document_metadata={"doc_id": "test_doc"}
        )

        for c1, c2 in zip(chunks1, chunks2, strict=False):
            pdq1 = c1.get("pdq_context")
            pdq2 = c2.get("pdq_context")

            if pdq1 is not None and pdq2 is not None:
                assert pdq1["policy"] == pdq2["policy"]
                assert pdq1["dimension"] == pdq2["dimension"]


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



================================================================================
FILE: tests/processing/test_semantic_chunking_properties.py
================================================================================

"""
Property-Based Tests for Semantic Chunking Determinism

Uses Hypothesis to verify that semantic chunking is deterministic, consistent,
and maintains correctness properties across different inputs.
"""

import hashlib
import re

import numpy as np
import pytest
from hypothesis import HealthCheck, assume, given, settings
from hypothesis import strategies as st

from src.farfan_pipeline.processing.semantic_chunking_policy import (
    BayesianEvidenceIntegrator,
    SemanticConfig,
    SemanticProcessor,
)


@st.composite
def spanish_policy_text(draw):
    """Generate realistic Spanish policy text for testing."""
    paragraphs = draw(st.lists(
        st.text(
            alphabet=st.characters(whitelist_categories=('L', 'N', 'P', 'Z'), max_codepoint=255),
            min_size=50,
            max_size=500
        ),
        min_size=1,
        max_size=20
    ))
    return "\n\n".join(p.strip() for p in paragraphs if p.strip())


@st.composite
def chunking_config_strategy(draw):
    """Generate valid SemanticConfig instances."""
    chunk_size = draw(st.integers(min_value=256, max_value=1024))
    chunk_overlap = draw(st.integers(min_value=32, max_value=chunk_size // 2))
    return SemanticConfig(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        similarity_threshold=draw(st.floats(min_value=0.5, max_value=0.95)),
        min_evidence_chunks=draw(st.integers(min_value=1, max_value=5)),
        bayesian_prior_strength=draw(st.floats(min_value=0.1, max_value=1.0)),
        device="cpu",
        batch_size=8,
        fp16=False
    )


class TestSemanticChunkingDeterminism:
    """Property-based tests for semantic chunking determinism."""

    @given(text=spanish_policy_text())
    @settings(max_examples=20, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunking_is_deterministic(self, text):
        """Property: Chunking the same text twice produces identical results."""
        assume(len(text) > 100)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks1 = processor.chunk_text(text, preserve_structure=False)
        chunks2 = processor.chunk_text(text, preserve_structure=False)

        assert len(chunks1) == len(chunks2), "Chunk count must be deterministic"

        for c1, c2 in zip(chunks1, chunks2, strict=False):
            assert c1["content"] == c2["content"], "Chunk content must be identical"
            assert c1["token_count"] == c2["token_count"], "Token counts must match"

    @given(text=spanish_policy_text())
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunks_preserve_all_text(self, text):
        """Property: All input text appears in output chunks (no loss)."""
        assume(len(text) > 100)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks = processor.chunk_text(text, preserve_structure=False)

        reconstructed = " ".join(c["content"] for c in chunks)

        important_words = re.findall(r'\b\w{4,}\b', text.lower())[:20]
        for word in important_words:
            assert word in reconstructed.lower(), f"Word '{word}' lost during chunking"

    @given(text=spanish_policy_text())
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_boundaries_are_coherent(self, text):
        """Property: Chunks don't split words or create malformed fragments."""
        assume(len(text) > 100)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks = processor.chunk_text(text, preserve_structure=False)

        for chunk in chunks:
            content = chunk["content"]
            assert content.strip() == content or len(content) < 10, \
                "Chunks should be trimmed or very short"

            assert not content.startswith(' ' * 5), "No excessive leading whitespace"
            assert not content.endswith(' ' * 5), "No excessive trailing whitespace"

    @given(
        text=spanish_policy_text(),
        config=chunking_config_strategy()
    )
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_sizes_respect_config(self, text, config):
        """Property: Chunks respect configured size limits."""
        assume(len(text) > config.chunk_size)

        processor = SemanticProcessor(config)
        chunks = processor.chunk_text(text, preserve_structure=False)
```

```python

        for chunk in chunks:
            token_count = chunk["token_count"]
            assert token_count <= config.chunk_size * 1.5, \
                f"Chunk exceeds size limit: {token_count} > {config.chunk_size * 1.5}"

    @given(text=spanish_policy_text())
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunks_have_required_fields(self, text):
        """Property: All chunks have required fields with correct types."""
        assume(len(text) > 100)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks = processor.chunk_text(text, preserve_structure=False)

        required_fields = ["content", "section_type", "token_count", "position",
                          "has_table", "has_numerical", "pdq_context"]

        for chunk in chunks:
            for field in required_fields:
                assert field in chunk, f"Missing required field: {field}"

            assert isinstance(chunk["content"], str)
            assert isinstance(chunk["token_count"], int)
            assert isinstance(chunk["position"], int)
            assert isinstance(chunk["has_table"], bool)
            assert isinstance(chunk["has_numerical"], bool)

    @given(text=spanish_policy_text())
    @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_embeddings_are_deterministic(self, text):
        """Property: Same text produces identical embeddings."""
        assume(len(text) > 50)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        emb1 = processor.embed_single(text)
        emb2 = processor.embed_single(text)

        np.testing.assert_array_almost_equal(emb1, emb2, decimal=5,
            err_msg="Embeddings must be deterministic")


class TestBayesianEvidenceProperties:
    """Property-based tests for Bayesian evidence integration."""

    @given(
        similarities=st.lists(st.floats(min_value=-1.0, max_value=1.0), min_size=1, max_size=20),
        prior_concentration=st.floats(min_value=0.1, max_value=2.0)
    )
    @settings(max_examples=50, deadline=1000)
    def test_evidence_integration_is_bounded(self, similarities, prior_concentration):
        """Property: Evidence integration produces scores in [0, 1]."""
        integrator = BayesianEvidenceIntegrator(prior_concentration)

        metadata = [{"position": i, "has_table": False, "has_numerical": False,
                     "section_type": "diagnostico"}
                    for i in range(len(similarities))]

        result = integrator.integrate_evidence(np.array(similarities), metadata)

        assert 0.0 <= result["posterior_mean"] <= 1.0, \
            f"Posterior mean out of bounds: {result['posterior_mean']}"
        assert 0.0 <= result["confidence"] <= 1.0, \
            f"Confidence out of bounds: {result['confidence']}"
        assert result["information_gain"] >= 0.0, \
            f"Information gain must be non-negative: {result['information_gain']}"

    @given(
        similarities=st.lists(st.floats(min_value=0.5, max_value=1.0), min_size=1, max_size=10)
    )
    @settings(max_examples=30, deadline=1000)
    def test_high_similarity_increases_confidence(self, similarities):
        """Property: Higher similarities lead to higher confidence."""
        integrator = BayesianEvidenceIntegrator(0.5)

        metadata = [{"position": i, "has_table": False, "has_numerical": False,
                     "section_type": "diagnostico"}
                    for i in range(len(similarities))]

        result = integrator.integrate_evidence(np.array(similarities), metadata)

        assert result["posterior_mean"] > 0.4, \
            "High similarities should produce high posterior mean"
        assert result["confidence"] > 0.3, \
            "High similarities should produce reasonable confidence"

    @given(
        n_chunks=st.integers(min_value=1, max_value=20),
        prior=st.floats(min_value=0.1, max_value=2.0)
    )
    @settings(max_examples=30, deadline=1000)
    def test_evidence_integration_is_symmetric(self, n_chunks, prior):
        """Property: Same similarities in different order produce same result."""
        integrator = BayesianEvidenceIntegrator(prior)

        similarities = np.random.RandomState(42).uniform(0.3, 0.9, n_chunks)
        metadata = [{"position": i, "has_table": False, "has_numerical": False,
                     "section_type": "diagnostico"}
                    for i in range(n_chunks)]

        result1 = integrator.integrate_evidence(similarities, metadata)

        shuffled_indices = np.random.RandomState(43).permutation(n_chunks)
        similarities_shuffled = similarities[shuffled_indices]
        metadata_shuffled = [metadata[i] for i in shuffled_indices]

        result2 = integrator.integrate_evidence(similarities_shuffled, metadata_shuffled)

        assert abs(result1["posterior_mean"] - result2["posterior_mean"]) < 0.15, \
            "Evidence integration should be approximately order-independent"


class TestSemanticChunkingInvariants:
    """Invariant tests for semantic chunking pipeline."""

    @given(text=spanish_policy_text())
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_positions_are_monotonic(self, text):
        """Property: Chunk positions increase monotonically."""
        assume(len(text) > 100)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks = processor.chunk_text(text, preserve_structure=False)

        positions = [c["position"] for c in chunks]
        assert positions == sorted(positions), "Chunk positions must be monotonic"

    @given(text=spanish_policy_text())
    @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_hashes_are_unique_for_different_content(self, text):
        """Property: Different chunk contents produce different hashes."""
        assume(len(text) > 200)

        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor = SemanticProcessor(config)

        chunks = processor.chunk_text(text, preserve_structure=False)

        if len(chunks) >= 2:
            hash1 = hashlib.sha256(chunks[0]["content"].encode()).hexdigest()
            hash2 = hashlib.sha256(chunks[1]["content"].encode()).hexdigest()

            if chunks[0]["content"] != chunks[1]["content"]:
                assert hash1 != hash2, "Different content must have different hashes"

    @given(
        text=spanish_policy_text(),
        seed=st.integers(min_value=0, max_value=1000)
    )
    @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunking_is_reproducible_with_seed(self, text, seed):
        """Property: Same seed produces same chunking."""
        assume(len(text) > 100)

        np.random.seed(seed)
        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor1 = SemanticProcessor(config)
        chunks1 = processor1.chunk_text(text, preserve_structure=False)

        np.random.seed(seed)
        config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
        processor2 = SemanticProcessor(config)
        chunks2 = processor2.chunk_text(text, preserve_structure=False)

        assert len(chunks1) == len(chunks2)
        for c1, c2 in zip(chunks1, chunks2, strict=False):
            assert c1["content"] == c2["content"]


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



================================================================================
FILE: tests/processing/test_spc_ingestion_properties.py
================================================================================

"""
Property-Based Tests for SPC Ingestion Pipeline Idempotency

Verifies that the SPC ingestion pipeline is idempotent, meaning running
it multiple times on the same input produces identical results.
"""

import hashlib
import json

import pytest
from hypothesis import HealthCheck, assume, given, settings
from hypothesis import strategies as st

from src.farfan_pipeline.processing.spc_ingestion import StrategicChunkingSystem


@st.composite
def minimal_policy_document(draw):
    """Generate minimal valid policy documents for testing."""
    title = draw(st.text(min_size=10, max_size=100))
    content_paragraphs = draw(st.lists(
        st.text(min_size=50, max_size=300),
        min_size=3,
        max_size=10
    ))
    content = "\n\n".join(content_paragraphs)

    return {
        "doc_id": draw(st.text(alphabet=st.characters(whitelist_categories=('L', 'N')),
                              min_size=5, max_size=20)),
        "title": title,
        "content": content
    }


class TestSPCIngestionIdempotency:
    """Property-based tests for SPC ingestion idempotency."""

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_ingestion_is_idempotent(self, doc):
        """Property: Running ingestion twice produces identical results."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        result1 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        result2 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        assert len(result1) == len(result2), "Chunk count must be identical"

        for i, (c1, c2) in enumerate(zip(result1, result2, strict=False)):
            c1_text = system.get_chunk_text(c1)
            c2_text = system.get_chunk_text(c2)
            assert c1_text == c2_text, f"Chunk {i} content differs"

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_hashes_are_stable(self, doc):
        """Property: Chunk hashes remain stable across runs."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        result1 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        result2 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        for c1, c2 in zip(result1, result2, strict=False):
            c1_text = system.get_chunk_text(c1)
            c2_text = system.get_chunk_text(c2)

            hash1 = hashlib.sha256(c1_text.encode()).hexdigest()
            hash2 = hashlib.sha256(c2_text.encode()).hexdigest()

            assert hash1 == hash2, "Chunk hashes must be stable"

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_metadata_is_consistent(self, doc):
        """Property: Chunk metadata is consistent across runs."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        result1 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        result2 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        for c1, c2 in zip(result1, result2, strict=False):
            meta1 = system.get_chunk_metadata(c1)
            meta2 = system.get_chunk_metadata(c2)

            assert meta1["document_id"] == meta2["document_id"]
            assert meta1["has_table"] == meta2["has_table"]
            assert meta1["has_list"] == meta2["has_list"]
            assert meta1["has_numbers"] == meta2["has_numbers"]

    @given(doc=minimal_policy_document())
    @settings(max_examples=8, deadline=15000, suppress_health_check=[HealthCheck.too_slow])
    def test_pdq_context_inference_is_stable(self, doc):
        """Property: PDQ context inference is stable across runs."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        result1 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        result2 = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        for c1, c2 in zip(result1, result2, strict=False):
            ctx1 = system.get_chunk_pdq_context(c1)
            ctx2 = system.get_chunk_pdq_context(c2)

            if ctx1 is not None and ctx2 is not None:
                assert ctx1["policy"] == ctx2["policy"]
                assert ctx1["dimension"] == ctx2["dimension"]


class TestSPCIngestionInvariants:
    """Invariant tests for SPC ingestion pipeline."""

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunks_cover_all_content(self, doc):
        """Property: Chunks collectively cover all important content."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        chunks = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        all_chunk_text = " ".join(system.get_chunk_text(c) for c in chunks)

        important_words = set()
        for word in doc["content"].split():
            if len(word) > 5:
                important_words.add(word.lower())

        coverage = sum(1 for word in important_words
                      if word in all_chunk_text.lower())

        coverage_ratio = coverage / max(len(important_words), 1)
        assert coverage_ratio > 0.7, \
            f"Chunks must cover at least 70% of important words: {coverage_ratio:.2%}"

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunks_have_minimum_quality(self, doc):
        """Property: All chunks meet minimum quality thresholds."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        chunks = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        for chunk in chunks:
            text = system.get_chunk_text(chunk)

            assert len(text) >= 20, f"Chunk too short: {len(text)} chars"

            assert text.strip(), "Chunk must not be empty"

            words = text.split()
            assert len(words) >= 5, f"Chunk has too few words: {len(words)}"

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_chunk_boundaries_dont_duplicate_content(self, doc):
        """Property: Chunks don’t have excessive duplication."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        chunks = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        if len(chunks) >= 2:
            for i in range(len(chunks) - 1):
                text1 = system.get_chunk_text(chunks[i])
                text2 = system.get_chunk_text(chunks[i + 1])

                words1 = set(text1.lower().split())
                words2 = set(text2.lower().split())

                if words1 and words2:
                    overlap = len(words1 & words2) / max(len(words1), len(words2))
                    assert overlap < 0.8, \
                        f"Excessive overlap between chunks {i} and {i+1}: {overlap:.2%}"


class TestSPCIngestionCorrectness:
    """Correctness properties for SPC ingestion."""

    @given(doc=minimal_policy_document())
    @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
    def test_document_id_propagates_to_chunks(self, doc):
        """Property: Document ID is preserved in all chunks."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        chunks = system.process_document(doc["content"], {
            "doc_id": doc["doc_id"],
            "title": doc["title"]
        })

        for chunk in chunks:
            meta = system.get_chunk_metadata(chunk)
            assert meta["document_id"] == doc["doc_id"], \
                "Document ID must propagate to all chunks"

    @given(
        doc=minimal_policy_document(),
        run_count=st.integers(min_value=2, max_value=4)
    )
    @settings(max_examples=5, deadline=20000, suppress_health_check=[HealthCheck.too_slow])
    def test_multiple_runs_produce_identical_output(self, doc, run_count):
        """Property: Multiple runs produce bit-for-bit identical output."""
        assume(len(doc["content"]) > 200)

        system = StrategicChunkingSystem()

        results = []
        for _ in range(run_count):
            chunks = system.process_document(doc["content"], {
                "doc_id": doc["doc_id"],
                "title": doc["title"]
            })

            serialized = json.dumps([
                {
                    "text": system.get_chunk_text(c),
                    "metadata": system.get_chunk_metadata(c)
                }
                for c in chunks
            ], sort_keys=True)

            results.append(hashlib.sha256(serialized.encode()).hexdigest())

        assert len(set(results)) == 1, \
            f"Multiple runs produced different outputs: {results}"


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



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




================================================================================
FILE: tests/synchronization/test_irrigation_synchronizer.py
================================================================================

"""Unit tests for ChunkMatrix validation and chunk routing."""

import sys
from datetime import datetime
from pathlib import Path

import pytest

project_root = Path(__file__).parent.parent.parent
sys.path.insert(0, str(project_root / "src"))

from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
    ChunkRoutingResult,
    IrrigationSynchronizer,
)
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from farfan_pipeline.synchronization.irrigation_synchronizer import ChunkMatrix


def create_chunk(
    chunk_id: int, policy_area_id: str, dimension_id: str, text: str = "test content"
) -> ChunkData:
    """Factory for creating test chunks."""
    return ChunkData(
        id=chunk_id,
        text=text,
        chunk_type="diagnostic",
        sentences=[],
        tables=[],
        start_pos=0,
        end_pos=len(text),
        confidence=0.95,
        policy_area_id=policy_area_id,
        dimension_id=dimension_id,
    )


def create_complete_document() -> PreprocessedDocument:
    """Create a valid document with all 60 required chunks."""
    chunks = []
    chunk_id = 0
    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunks.append(create_chunk(chunk_id, pa_id, dim_id))
            chunk_id += 1

    return PreprocessedDocument(
        document_id="test-doc",
        raw_text="test",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
        ingested_at=datetime.now(),
    )


def test_chunk_matrix_accepts_valid_60_chunks() -> None:
    """ChunkMatrix should accept exactly 60 valid chunks."""
    doc = create_complete_document()
    matrix = ChunkMatrix(doc)

    chunk = matrix.get_chunk("PA01", "DIM01")
    assert chunk.policy_area_id == "PA01"
    assert chunk.dimension_id == "DIM01"


def test_chunk_matrix_rejects_59_chunks() -> None:
    """ChunkMatrix should reject documents with 59 chunks by detecting missing combination."""
    doc = create_complete_document()
    doc.chunks = doc.chunks[:59]

    with pytest.raises(ValueError) as exc_info:
        ChunkMatrix(doc)

    error_msg = str(exc_info.value)
    assert "Missing chunk combinations" in error_msg


def test_chunk_matrix_rejects_61_chunks() -> None:
    """ChunkMatrix should reject documents with 61 chunks by detecting duplicate."""
    doc = create_complete_document()
    duplicate_chunk = doc.chunks[0]
    doc.chunks.append(
        create_chunk(
            60,
            duplicate_chunk.policy_area_id,
            duplicate_chunk.dimension_id,
            "duplicate",
        )
    )

    with pytest.raises(ValueError) as exc_info:
        ChunkMatrix(doc)

    error_msg = str(exc_info.value)
    assert "Duplicate (PA, DIM) combination detected" in error_msg


def test_chunk_matrix_rejects_duplicate_keys() -> None:
    """ChunkMatrix should reject documents with duplicate PA-DIM combinations."""
    chunks = []
    chunks.append(create_chunk(0, "PA01", "DIM01"))
    chunks.append(create_chunk(1, "PA01", "DIM01"))

    chunk_id = 2
    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            if pa_num == 1 and dim_num == 1:
                continue
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunks.append(create_chunk(chunk_id, pa_id, dim_id))
            chunk_id += 1

    doc = PreprocessedDocument(
        document_id="test-doc",
        raw_text="test",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
        ingested_at=datetime.now(),
    )

    with pytest.raises(ValueError, match=r"Duplicate .*PA01-DIM01"):
        ChunkMatrix(doc)


def test_chunk_matrix_detects_missing_combinations() -> None:
    """ChunkMatrix should detect and report missing PA-DIM combinations with 59 chunks."""
    chunks = []
    chunk_id = 0
    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            if pa_num == 5 and dim_num == 3:
                continue
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunks.append(create_chunk(chunk_id, pa_id, dim_id))
            chunk_id += 1

    doc = PreprocessedDocument(
        document_id="test-doc",
        raw_text="test",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
        ingested_at=datetime.now(),
    )

    with pytest.raises(ValueError) as exc_info:
        ChunkMatrix(doc)

    error_msg = str(exc_info.value)
    assert "Missing chunk combinations" in error_msg
    assert "PA05', 'DIM03" in error_msg


def test_chunk_matrix_detects_missing_combinations_with_exact_error() -> None:
    """ChunkMatrix should report missing PA-DIM combination when 60 chunks but duplicate."""
    doc = create_complete_document()
    original_chunk = doc.chunks[0]
    doc.chunks[0] = create_chunk(
        0, original_chunk.policy_area_id, original_chunk.dimension_id
    )
    doc.chunks[10] = create_chunk(
        10, original_chunk.policy_area_id, original_chunk.dimension_id
    )

    with pytest.raises(ValueError) as exc_info:
        ChunkMatrix(doc)

    error_msg = str(exc_info.value)
    assert "Duplicate" in error_msg
    assert f"{original_chunk.policy_area_id}-{original_chunk.dimension_id}" in error_msg


def test_chunk_matrix_reports_specific_missing_key() -> None:
    """ChunkMatrix should report specific missing key PA05-DIM03 in error message."""
    chunks = []
    chunk_id = 0
    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            if pa_num == 5 and dim_num == 3:
                chunk_id += 1
                continue
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunks.append(create_chunk(chunk_id, pa_id, dim_id))
            chunk_id += 1

    chunks.append(create_chunk(59, "PA05", "DIM03"))

    doc = PreprocessedDocument(
        document_id="test-doc",
        raw_text="test",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
        ingested_at=datetime.now(),
    )

    matrix = ChunkMatrix(doc)
    chunk = matrix.get_chunk("PA05", "DIM03")
    assert chunk.policy_area_id == "PA05"
    assert chunk.dimension_id == "DIM03"


def test_chunk_matrix_rejects_null_policy_area_id() -> None:
    """ChunkData should reject chunks with null policy_area_id at creation time."""
    with pytest.raises(ValueError, match="chunk_id is required"):
        ChunkData(
            id=0,
            text="test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=4,
            confidence=0.95,
            policy_area_id=None,
            dimension_id="DIM01",
        )


def test_chunk_matrix_rejects_null_dimension_id() -> None:
    """ChunkData should reject chunks with null dimension_id at creation time."""
    with pytest.raises(ValueError, match="chunk_id is required"):
        ChunkData(
            id=0,
            text="test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=4,
            confidence=0.95,
            policy_area_id="PA01",
            dimension_id=None,
        )


def test_chunk_matrix_rejects_invalid_chunk_id_format_pa() -> None:
    """ChunkData should reject invalid policy area format at creation time."""
    with pytest.raises(ValueError, match=r"Invalid chunk_id"):
        create_chunk(0, "PA11", "DIM01")


def test_chunk_matrix_rejects_invalid_chunk_id_format_dim() -> None:
    """ChunkData should reject invalid dimension format at creation time."""
    with pytest.raises(ValueError, match=r"Invalid chunk_id"):
        create_chunk(0, "PA01", "DIM07")


def test_chunk_matrix_rejects_malformed_chunk_id() -> None:
    """ChunkData should reject malformed chunk IDs at creation time."""
    with pytest.raises(ValueError, match=r"Invalid chunk_id"):
        create_chunk(0, "P01", "DIM01")


def test_chunk_matrix_get_chunk_success() -> None:
    """get_chunk should return correct chunk for valid keys."""
    doc = create_complete_document()
    matrix = ChunkMatrix(doc)

    chunk = matrix.get_chunk("PA05", "DIM03")
    assert chunk.policy_area_id == "PA05"
    assert chunk.dimension_id == "DIM03"


def test_chunk_matrix_get_chunk_all_combinations() -> None:
    """get_chunk should work for all 60 valid combinations."""
    doc = create_complete_document()
    matrix = ChunkMatrix(doc)

    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunk = matrix.get_chunk(pa_id, dim_id)
            assert chunk.policy_area_id == pa_id
            assert chunk.dimension_id == dim_id


def test_chunk_matrix_get_chunk_missing_key() -> None:
    """get_chunk should raise KeyError for missing combinations."""
    doc = create_complete_document()
    doc.chunks = [
        chunk
        for chunk in doc.chunks
        if not (chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03")
    ]
    doc.chunks.append(create_chunk(59, "PA05", "DIM03"))

    matrix = ChunkMatrix(doc)

    with pytest.raises(KeyError, match="Chunk not found for key: PA11-DIM01"):
        matrix.get_chunk("PA11", "DIM01")


def test_chunk_matrix_get_chunk_o1_lookup() -> None:
    """get_chunk should provide O(1) lookup performance."""
    doc = create_complete_document()
    matrix = ChunkMatrix(doc)

    import time

    iterations = 1000
    start = time.perf_counter()
    for _ in range(iterations):
        matrix.get_chunk("PA05", "DIM03")
    elapsed = time.perf_counter() - start

    assert elapsed < 0.1, f"O(1) lookup too slow: {elapsed}s for {iterations} lookups"


def test_chunk_matrix_preserves_chunk_content() -> None:
    """ChunkMatrix should preserve original chunk content and metadata."""
    doc = create_complete_document()
    doc.chunks[0] = ChunkData(
        id=42,
        text="specific test content",
        chunk_type="activity",
        sentences=[1, 2, 3],
        tables=[],
        start_pos=10,
        end_pos=31,
        confidence=0.87,
        policy_area_id="PA01",
        dimension_id="DIM01",
    )

    matrix = ChunkMatrix(doc)
    chunk = matrix.get_chunk("PA01", "DIM01")

    assert chunk.id == 42
    assert chunk.text == "specific test content"
    assert chunk.chunk_type == "activity"
    assert chunk.sentences == [1, 2, 3]
    assert chunk.confidence == 0.87


def test_validate_chunk_routing_success() -> None:
    """validate_chunk_routing should successfully route valid question to chunk."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {
        "question_id": "D1-Q01",
        "policy_area_id": "PA05",
        "dimension_id": "DIM03",
        "question_text": "Test question",
    }

    result = synchronizer.validate_chunk_routing(question)

    assert isinstance(result, ChunkRoutingResult)
    assert result.chunk_id == "PA05-DIM03"
    assert result.policy_area_id == "PA05"
    assert result.dimension_id == "DIM03"
    assert result.text_content == "test content"
    assert result.target_chunk.policy_area_id == "PA05"
    assert result.target_chunk.dimension_id == "DIM03"


def test_validate_chunk_routing_missing_question_id() -> None:
    """validate_chunk_routing should raise ValueError if question_id missing."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {"policy_area_id": "PA05", "dimension_id": "DIM03"}

    with pytest.raises(ValueError, match="missing required 'question_id' field"):
        synchronizer.validate_chunk_routing(question)


def test_validate_chunk_routing_missing_policy_area_id() -> None:
    """validate_chunk_routing should raise ValueError if policy_area_id missing."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {"question_id": "D1-Q01", "dimension_id": "DIM03"}

    with pytest.raises(ValueError, match="missing required 'policy_area_id' field"):
        synchronizer.validate_chunk_routing(question)


def test_validate_chunk_routing_missing_dimension_id() -> None:
    """validate_chunk_routing should raise ValueError if dimension_id missing."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {"question_id": "D1-Q01", "policy_area_id": "PA05"}

    with pytest.raises(ValueError, match="missing required 'dimension_id' field"):
        synchronizer.validate_chunk_routing(question)


def test_validate_chunk_routing_chunk_not_found() -> None:
    """validate_chunk_routing should raise ValueError if chunk not in matrix."""
    doc = create_complete_document()
    doc.chunks = [
        chunk
        for chunk in doc.chunks
        if not (chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03")
    ]

    questionnaire = {"blocks": {}}

    with pytest.raises(
        ValueError,
        match="Chunk matrix validation failed during synchronizer initialization",
    ):
        IrrigationSynchronizer(questionnaire=questionnaire, preprocessed_document=doc)


def test_validate_chunk_routing_empty_text() -> None:
    """validate_chunk_routing should raise ValueError if chunk text is empty."""
    doc = create_complete_document()
    for chunk in doc.chunks:
        if chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03":
            idx = doc.chunks.index(chunk)
            doc.chunks[idx] = ChunkData(
                id=chunk.id,
                text="",
                chunk_type=chunk.chunk_type,
                sentences=chunk.sentences,
                tables=chunk.tables,
                start_pos=chunk.start_pos,
                end_pos=chunk.end_pos,
                confidence=chunk.confidence,
                policy_area_id=chunk.policy_area_id,
                dimension_id=chunk.dimension_id,
            )
            break

    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {
        "question_id": "D1-Q01",
        "policy_area_id": "PA05",
        "dimension_id": "DIM03",
    }

    with pytest.raises(ValueError, match="empty or whitespace-only text"):
        synchronizer.validate_chunk_routing(question)


def test_validate_chunk_routing_whitespace_only_text() -> None:
    """validate_chunk_routing should raise ValueError if chunk text is whitespace."""
    doc = create_complete_document()
    for chunk in doc.chunks:
        if chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03":
            idx = doc.chunks.index(chunk)
            doc.chunks[idx] = ChunkData(
                id=chunk.id,
                text="   \n\t  ",
                chunk_type=chunk.chunk_type,
                sentences=chunk.sentences,
                tables=chunk.tables,
                start_pos=chunk.start_pos,
                end_pos=chunk.end_pos,
                confidence=chunk.confidence,
                policy_area_id=chunk.policy_area_id,
                dimension_id=chunk.dimension_id,
            )
            break

    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {
        "question_id": "D1-Q01",
        "policy_area_id": "PA05",
        "dimension_id": "DIM03",
    }

    with pytest.raises(ValueError, match="empty or whitespace-only text"):
        synchronizer.validate_chunk_routing(question)


def test_validate_chunk_routing_expected_elements_default() -> None:
    """validate_chunk_routing should handle missing expected_elements gracefully."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {
        "question_id": "D1-Q01",
        "policy_area_id": "PA05",
        "dimension_id": "DIM03",
    }

    result = synchronizer.validate_chunk_routing(question)

    assert result.expected_elements == []


def test_validate_chunk_routing_immutability() -> None:
    """ChunkRoutingResult should be immutable."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    question = {
        "question_id": "D1-Q01",
        "policy_area_id": "PA05",
        "dimension_id": "DIM03",
    }

    result = synchronizer.validate_chunk_routing(question)

    with pytest.raises(Exception):
        result.chunk_id = "different_id"


def test_filter_patterns_with_matching_policy_area() -> None:
    """_filter_patterns should include only patterns with matching policy_area_id."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    patterns = [
        {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
        {"id": "PAT-002", "policy_area_id": "PA02", "pattern": "test2"},
        {"id": "PAT-003", "policy_area_id": "PA01", "pattern": "test3"},
        {"id": "PAT-004", "pattern": "test4"},
    ]

    result = synchronizer._filter_patterns(patterns, "PA01")

    assert isinstance(result, tuple)
    assert len(result) == 2
    assert result[0]["id"] == "PAT-001"
    assert result[1]["id"] == "PAT-003"


def test_filter_patterns_excludes_without_policy_area_id() -> None:
    """_filter_patterns should exclude patterns without policy_area_id attribute."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    patterns = [
        {"id": "PAT-001", "pattern": "test1"},
        {"id": "PAT-002", "pattern": "test2"},
    ]

    result = synchronizer._filter_patterns(patterns, "PA01")

    assert isinstance(result, tuple)
    assert len(result) == 0


def test_filter_patterns_returns_empty_tuple_when_no_matches() -> None:
    """_filter_patterns should return empty tuple when no patterns match."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    patterns = [
        {"id": "PAT-001", "policy_area_id": "PA02", "pattern": "test1"},
        {"id": "PAT-002", "policy_area_id": "PA03", "pattern": "test2"},
    ]

    result = synchronizer._filter_patterns(patterns, "PA01")

    assert isinstance(result, tuple)
    assert len(result) == 0


def test_filter_patterns_immutability() -> None:
    """_filter_patterns should return an immutable tuple."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    patterns = [
        {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
    ]

    result = synchronizer._filter_patterns(patterns, "PA01")

    assert isinstance(result, tuple)
    with pytest.raises(AttributeError):
        result.append({"id": "PAT-002", "pattern": "test2"})


def test_filter_patterns_strict_equality() -> None:
    """_filter_patterns should use strict equality for policy_area_id matching."""
    doc = create_complete_document()
    questionnaire = {"blocks": {}}
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire, preprocessed_document=doc
    )

    patterns = [
        {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
        {"id": "PAT-002", "policy_area_id": "PA010", "pattern": "test2"},
        {"id": "PAT-003", "policy_area_id": "PA1", "pattern": "test3"},
    ]

    result = synchronizer._filter_patterns(patterns, "PA01")

    assert isinstance(result, tuple)
    assert len(result) == 1
    assert result[0]["id"] == "PAT-001"

```

```


================================================================================
FILE: tests/test_aggregation_sota.py
================================================================================

"""
Tests for SOTA aggregation features.

This module tests the three SOTA injection points:
1. Choquet integral aggregation (non-linear interactions)
2. Bayesian uncertainty quantification (bootstrap + CI)
3. DAG provenance tracking (lineage + SHAP attribution)
"""

import pytest
import numpy as np

from farfan_pipeline.processing.aggregation_provenance import (
    AggregationDAG,
    ProvenanceNode,
)
from farfan_pipeline.processing.uncertainty_quantification import (
    BootstrapAggregator,
    BayesianPropagation,
    SensitivityAnalysis,
    aggregate_with_uncertainty,
)
from farfan_pipeline.processing.choquet_adapter import (
    ChoquetProcessingAdapter,
    create_default_choquet_adapter,
)


class TestChoquetAggregation:
    """Test Choquet integral aggregation."""

    def test_choquet_creates_non_linear_interactions(self):
        """Choquet with interactions should differ from naive weighted average."""
        scores = [7.0, 8.0, 6.5]
        weights = [0.33, 0.33, 0.34]

        # Naive weighted average
        naive_score = sum(s * w for s, w in zip(scores, weights))

        # Choquet with interactions
        interaction_weights = {("0", "1"): 0.1}  # Synergy between first two
        choquet = ChoquetProcessingAdapter(
            linear_weights={str(i): w for i, w in enumerate(weights)},
            interaction_weights=interaction_weights,
        )
        choquet_score = choquet.aggregate(scores, weights, interaction_pairs=[(0, 1)])

        # Choquet should differ due to min(7.0, 8.0) * 0.1 = 0.7 interaction term
        assert abs(choquet_score - naive_score) > 0.01
        assert choquet_score > naive_score  # Positive interaction (synergy)

    def test_choquet_with_redundancy_penalty(self):
        """Negative interaction weights should penalize redundancy."""
        scores = [8.0, 8.0]  # Redundant high scores
        weights = [0.5, 0.5]

        interaction_weights = {("0", "1"): -0.1}  # Redundancy penalty
        choquet = ChoquetProcessingAdapter(
            linear_weights={"0": 0.5, "1": 0.5},
            interaction_weights=interaction_weights,
        )
        choquet_score = choquet.aggregate(scores, weights, interaction_pairs=[(0, 1)])

        naive_score = 8.0  # Weighted average
        assert choquet_score < naive_score  # Penalty applied

    def test_choquet_default_adapter(self):
        """Default adapter should work for n inputs."""
        adapter = create_default_choquet_adapter(n_inputs=5)
        scores = [7.0, 8.0, 6.5, 7.5, 8.5]
        result = adapter.aggregate(scores, weights=None)

        assert 0.0 <= result <= 10.0  # Reasonable range
        assert isinstance(result, float)


class TestUncertaintyQuantification:
    """Test Bayesian uncertainty quantification."""

    def test_bootstrap_computes_confidence_intervals(self):
        """Bootstrap should provide 95% CI around mean."""
        scores = [7.0, 7.5, 6.8, 7.2, 6.9]
        weights = [0.2, 0.2, 0.2, 0.2, 0.2]

        bootstrapper = BootstrapAggregator(n_samples=1000, random_seed=42)
        uncertainty = bootstrapper.bootstrap_weighted_average(scores, weights)

        # Mean should be close to weighted average
        expected_mean = sum(s * w for s, w in zip(scores, weights))
        assert abs(uncertainty.mean - expected_mean) < 0.1

        # CI should contain mean
        lower, upper = uncertainty.confidence_interval_95
        assert lower < uncertainty.mean < upper

        # Std should be positive
        assert uncertainty.std > 0.0

    def test_bootstrap_is_deterministic(self):
        """Same seed should produce identical results."""
        scores = [7.0, 8.0, 6.5]

        bootstrap1 = BootstrapAggregator(n_samples=500, random_seed=42)
        result1 = bootstrap1.bootstrap_weighted_average(scores)

        bootstrap2 = BootstrapAggregator(n_samples=500, random_seed=42)
        result2 = bootstrap2.bootstrap_weighted_average(scores)

        assert result1.mean == result2.mean
        assert result1.std == result2.std
        assert result1.confidence_interval_95 == result2.confidence_interval_95

    def test_uncertainty_decomposition(self):
        """Epistemic and aleatoric uncertainty should be non-negative."""
        scores = [7.0, 8.0, 6.5, 7.5, 8.2]

        _, uncertainty = aggregate_with_uncertainty(scores, n_bootstrap=1000, random_seed=42)

        assert uncertainty.epistemic_uncertainty >= 0.0
        assert uncertainty.aleatoric_uncertainty >= 0.0
        assert uncertainty.dominant_uncertainty_type() in ["epistemic", "aleatoric", "balanced"]

    def test_bayesian_propagation_analytical(self):
        """Analytical propagation should match expected variance formula."""
        scores = [7.0, 8.0, 6.0]
        uncertainties = [0.5, 0.3, 0.4]
        weights = [0.33, 0.33, 0.34]

        mean, std = BayesianPropagation.propagate_weighted_average(
            scores, uncertainties, weights
        )

        # Mean should match weighted average
        expected_mean = sum(s * w for s, w in zip(scores, weights))
        assert abs(mean - expected_mean) < 1e-9

        # Variance: sum(w_i^2 * var_i)
        expected_var = sum(w**2 * u**2 for w, u in zip(weights, uncertainties))
        expected_std = np.sqrt(expected_var)
        assert abs(std - expected_std) < 1e-9

    def test_coefficient_of_variation_threshold(self):
        """High CV should trigger high uncertainty flag."""
        # Low uncertainty case
        scores_low_var = [7.0, 7.1, 7.05, 6.95, 7.0]
        _, unc_low = aggregate_with_uncertainty(scores_low_var, n_bootstrap=1000, random_seed=42)
        assert not unc_low.is_high_uncertainty(threshold=0.1)

        # High uncertainty case
        scores_high_var = [3.0, 9.0, 2.0, 8.5, 4.0]
        _, unc_high = aggregate_with_uncertainty(scores_high_var, n_bootstrap=1000, random_seed=42)
        assert unc_high.is_high_uncertainty(threshold=0.1)


class TestProvenanceDAG:
    """Test DAG provenance tracking."""

    def test_dag_creation_and_node_addition(self):
        """DAG should accept nodes and maintain structure."""
        dag = AggregationDAG()

        node1 = ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO")
        node2 = ProvenanceNode(node_id="Q002", level="micro", score=8.0, quality_level="EXCELENTE")

        dag.add_node(node1)
        dag.add_node(node2)

        assert "Q001" in dag.nodes
        assert "Q002" in dag.nodes
        assert dag.graph.number_of_nodes() == 2

    def test_dag_aggregation_edges(self):
        """DAG should record aggregation operations."""
        dag = AggregationDAG()

        # Add source nodes
        for i in range(3):
            node = ProvenanceNode(
                node_id=f"Q{i:03d}",
                level="micro",
                score=7.0 + i,
                quality_level="BUENO",
            )
            dag.add_node(node)

        # Add target node
        target = ProvenanceNode(
            node_id="DIM01_PA01",
            level="dimension",
            score=7.5,
            quality_level="BUENO",
        )
        dag.add_node(target)

        # Record aggregation
        dag.add_aggregation_edge(
            source_ids=["Q000", "Q001", "Q002"],
            target_id="DIM01_PA01",
            operation="weighted_average",
            weights=[0.33, 0.33, 0.34],
        )

        assert dag.graph.number_of_edges() == 3
        assert dag.graph.has_edge("Q000", "DIM01_PA01")

    def test_dag_cycle_detection(self):
        """DAG should reject edges that create cycles."""
        dag = AggregationDAG()

        dag.add_node(ProvenanceNode(node_id="A", level="micro", score=7.0, quality_level="BUENO"))
        dag.add_node(ProvenanceNode(node_id="B", level="dimension", score=7.5, quality_level="BUENO"))

        # A â\206\222 B is valid
        dag.add_aggregation_edge(["A"], "B", "weighted_average", [1.0])

        # B â\206\222 A would create cycle, should raise
        with pytest.raises(ValueError, match="cycle"):
            dag.add_aggregation_edge(["B"], "A", "weighted_average", [1.0])

    def test_lineage_tracing(self):
        """Should trace complete lineage of a target node."""
        dag = AggregationDAG()

        # Build hierarchy: Q001, Q002 â\206\222 DIM01 â\206\222 AREA01
        for qid in ["Q001", "Q002"]:
            dag.add_node(ProvenanceNode(node_id=qid, level="micro", score=7.0, quality_level="BUENO"))

        dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.5, quality_level="BUENO"))
        dag.add_aggregation_edge(["Q001", "Q002"], "DIM01", "weighted_average", [0.5, 0.5])

        dag.add_node(ProvenanceNode(node_id="AREA01", level="area", score=7.5, quality_level="BUENO"))
        dag.add_aggregation_edge(["DIM01"], "AREA01", "weighted_average", [1.0])

        # Trace AREA01
        lineage = dag.trace_lineage("AREA01")

        assert lineage["ancestor_count"] == 3  # Q001, Q002, DIM01
        assert "Q001" in lineage["ancestors"]
        assert "Q002" in lineage["ancestors"]
        assert "DIM01" in lineage["ancestors"]
        assert lineage["micro_question_count"] == 2
        # depth can be fractional due to average path lengths, just check > 1
        assert lineage["depth"] >= 1

    def test_shapley_attribution(self):
        """Shapley values should sum to target score."""
        dag = AggregationDAG()

        # Add sources
        scores = [7.0, 8.0, 6.5]
        weights = [0.3, 0.4, 0.3]
        for i, score in enumerate(scores):
            dag.add_node(ProvenanceNode(
                node_id=f"Q{i:03d}",
                level="micro",
                score=score,
                quality_level="BUENO",
            ))

        # Add target
        target_score = sum(s * w for s, w in zip(scores, weights))
        dag.add_node(ProvenanceNode(
            node_id="DIM01",
            level="dimension",
            score=target_score,
            quality_level="BUENO",
        ))

        dag.add_aggregation_edge(
            ["Q000", "Q001", "Q002"],
            "DIM01",
            "weighted_average",
            weights,
        )

        # Compute Shapley
        attribution = dag.compute_shapley_attribution("DIM01")

        # Shapley values should sum to target score (approximately)
        total_attribution = sum(attribution.values())
        assert abs(total_attribution - target_score) < 0.01

        # Highest weight should have highest attribution
        max_weight_idx = weights.index(max(weights))
        max_attr_node = max(attribution, key=attribution.get)
        assert max_attr_node == f"Q{max_weight_idx:03d}"

    def test_critical_path_identification(self):
        """Should identify top-k most important inputs."""
        dag = AggregationDAG()

        scores = [9.0, 7.0, 8.0, 6.0, 7.5]
        weights = [0.3, 0.2, 0.25, 0.1, 0.15]

        for i, score in enumerate(scores):
            dag.add_node(ProvenanceNode(
                node_id=f"Q{i:03d}",
                level="micro",
                score=score,
                quality_level="BUENO",
            ))

        target_score = sum(s * w for s, w in zip(scores, weights))
        dag.add_node(ProvenanceNode(
            node_id="DIM01",
            level="dimension",
            score=target_score,
            quality_level="BUENO",
        ))

        dag.add_aggregation_edge(
            [f"Q{i:03d}" for i in range(5)],
            "DIM01",
            "weighted_average",
            weights,
        )

        critical_path = dag.get_critical_path("DIM01", top_k=3)

        assert len(critical_path) == 3
        # Top contributor should be Q000 (highest weight * highest score)
        assert critical_path[0][0] == "Q000"

    def test_graphml_export(self, tmp_path):
        """Should export DAG to GraphML format."""
        dag = AggregationDAG()

        dag.add_node(ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO"))
        dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.0, quality_level="BUENO"))
        dag.add_aggregation_edge(["Q001"], "DIM01", "weighted_average", [1.0])

        output_path = tmp_path / "test_dag.graphml"
        dag.export_graphml(str(output_path))

        assert output_path.exists()
        # Verify it's valid XML
        with open(output_path) as f:
            content = f.read()
            assert "<?xml" in content
            assert "<graphml" in content

    def test_prov_json_export(self, tmp_path):
        """Should export to W3C PROV-JSON format."""
        dag = AggregationDAG()

        dag.add_node(ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO"))
        dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.0, quality_level="BUENO"))
        dag.add_aggregation_edge(["Q001"], "DIM01", "weighted_average", [1.0])

        output_path = tmp_path / "provenance.json"
        dag.export_prov_json(str(output_path))

        assert output_path.exists()

        import json
        with open(output_path) as f:
            prov_doc = json.load(f)

        assert "prefix" in prov_doc
        assert "entity" in prov_doc
        assert "activity" in prov_doc
        assert "Q001" in prov_doc["entity"]

    def test_dag_statistics(self):
        """Should compute graph statistics."""
        dag = AggregationDAG()

        for i in range(5):
            dag.add_node(ProvenanceNode(
                node_id=f"Q{i:03d}",
                level="micro",
                score=7.0,
                quality_level="BUENO",
            ))

        dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.5, quality_level="BUENO"))
        dag.add_aggregation_edge(
            [f"Q{i:03d}" for i in range(5)],
            "DIM01",
            "weighted_average",
            [0.2] * 5,
        )

        stats = dag.get_statistics()

        assert stats["node_count"] == 6
        assert stats["edge_count"] == 5
        assert stats["is_dag"] is True
        assert stats["nodes_by_level"]["micro"] == 5
        assert stats["nodes_by_level"]["dimension"] == 1


class TestSensitivityAnalysis:
    """Test Sobol sensitivity analysis."""

    def test_sobol_indices_sum_to_one(self):
        """First-order Sobol indices should sum to ˜1.0."""
        scores = [7.0, 8.0, 6.5, 7.5]
        weights = [0.25, 0.25, 0.25, 0.25]

        sobol = SensitivityAnalysis.compute_sobol_indices(
            scores, weights, n_samples=500, random_seed=42
        )

        total = sum(sobol.values())
        assert abs(total - 1.0) < 0.05  # Allow 5% tolerance for MC error

    def test_sobol_identifies_influential_inputs(self):
        """Inputs with high variance should have high Sobol index."""
        # Input 0 has high weight but low variance
        # Input 1 has medium weight and high variance (will be perturbed more)
        scores = [7.0, 7.0, 7.0]
        weights = [0.5, 0.3, 0.2]

        sobol = SensitivityAnalysis.compute_sobol_indices(
            scores, weights, n_samples=500, random_seed=42
        )

        # All have same score, so sensitivity is proportional to weight^2
        # (for weighted average, Sobol ˜ w^2 * var, but var is introduced by MC)
        assert len(sobol) == 3


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



================================================================================
FILE: tests/test_batch_executor.py
================================================================================

"""Tests for batch executor infrastructure."""

from pathlib import Path
from unittest.mock import MagicMock

import pytest

from farfan_pipeline.core.orchestrator.batch_executor import (
    COMPLEXITY_BATCH_SIZE_MAP,
    EXECUTOR_COMPLEXITY_MAP,
    AggregatedBatchResults,
    BatchExecutor,
    BatchExecutorConfig,
    BatchResult,
    BatchStatus,
    ExecutorComplexity,
)
from farfan_pipeline.core.types import PreprocessedDocument


@pytest.fixture
def batch_config():
    """Create batch executor configuration."""
    return BatchExecutorConfig(
        default_batch_size=10,
        max_batch_size=50,
        min_batch_size=2,
        enable_streaming=True,
        error_threshold=0.5,
        max_retries=2,
        enable_instrumentation=False,
    )


@pytest.fixture
def mock_method_executor():
    """Create mock method executor."""
    executor = MagicMock()
    executor.signal_registry = MagicMock()
    return executor


@pytest.fixture
def sample_document():
    """Create sample preprocessed document."""
    return PreprocessedDocument(
        document_id="test_doc_001",
        raw_text="Sample document text for testing batch processing.",
        sentences=["Sample document text for testing batch processing."],
        tables=[],
        metadata={"source": "test"},
        source_path=Path("/tmp/test.pdf"),
    )


@pytest.fixture
def sample_question_context():
    """Create sample question context."""
    return {
        "question_id": "D1-Q1",
        "question_global": 1,
        "base_slot": "D1-Q1",
        "dimension_id": "DIM01",
        "policy_area_id": "PA01",
        "cluster_id": "CLU01",
        "scoring_modality": "quantitative",
        "expected_elements": ["baseline", "metrics"],
    }


class MockExecutor:
    """Mock executor for testing."""

    def __init__(
        self,
        method_executor,
        signal_registry,
        config,
        questionnaire_provider,
        calibration_orchestrator,
    ):
        self.method_executor = method_executor
        self.signal_registry = signal_registry
        self.config = config
        self.questionnaire_provider = questionnaire_provider
        self.calibration_orchestrator = calibration_orchestrator
        self.call_count = 0

    def execute(self, document, method_executor, question_context):
        """Mock execute method."""
        self.call_count += 1
        return {
            "evidence": {"baseline": "test_value"},
            "metadata": {"executed": True},
            "call_count": self.call_count,
        }


class FailingExecutor:
    """Mock executor that fails."""

    def __init__(
        self,
        method_executor,
        signal_registry,
        config,
        questionnaire_provider,
        calibration_orchestrator,
    ):
        self.method_executor = method_executor
        self.signal_registry = signal_registry

    def execute(self, document, method_executor, question_context):
        """Mock execute that always fails."""
        raise ValueError("Intentional test failure")


class PartialFailExecutor:
    """Mock executor that fails every other call."""

    def __init__(
        self,
        method_executor,
        signal_registry,
        config,
        questionnaire_provider,
        calibration_orchestrator,
    ):
        self.method_executor = method_executor
        self.call_count = 0

    def execute(self, document, method_executor, question_context):
        """Mock execute that fails every other call."""
        self.call_count += 1
        if self.call_count % 2 == 0:
            raise ValueError(f"Intentional failure on call {self.call_count}")
        return {"evidence": {"data": f"success_{self.call_count}"}}


class TestBatchExecutorConfig:
    """Test batch executor configuration."""

    def test_valid_config(self):
        """Test valid configuration."""
        config = BatchExecutorConfig(
            default_batch_size=10,
            max_batch_size=100,
            min_batch_size=1,
            error_threshold=0.3,
        )
        assert config.default_batch_size == 10
        assert config.max_batch_size == 100
        assert config.min_batch_size == 1
        assert config.error_threshold == 0.3

    def test_invalid_batch_sizes(self):
        """Test invalid batch size configuration."""
        with pytest.raises(ValueError, match="Invalid batch size configuration"):
            BatchExecutorConfig(
                default_batch_size=5,
                max_batch_size=3,
                min_batch_size=1,
            )

    def test_invalid_error_threshold(self):
        """Test invalid error threshold."""
        with pytest.raises(ValueError, match="error_threshold must be in"):
            BatchExecutorConfig(error_threshold=1.5)

    def test_default_values(self):
        """Test default configuration values."""
        config = BatchExecutorConfig()
        assert config.default_batch_size == 10
        assert config.max_batch_size == 100
        assert config.min_batch_size == 1
        assert config.error_threshold == 0.5
        assert config.max_retries == 2
        assert config.enable_streaming is True
```

```


class TestComplexityMapping:
    """Test executor complexity mapping."""

    def test_complexity_map_coverage(self):
        """Test that all major executors have complexity mappings."""
        assert "D1-Q1" in EXECUTOR_COMPLEXITY_MAP
        assert "D3-Q2" in EXECUTOR_COMPLEXITY_MAP
        assert "D5-Q2" in EXECUTOR_COMPLEXITY_MAP

    def test_complexity_batch_size_map(self):
        """Test complexity to batch size mapping."""
        assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.SIMPLE] == 50
        assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.MODERATE] == 20
        assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.COMPLEX] == 10
        assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.VERY_COMPLEX] == 5

    def test_very_complex_executors(self):
        """Test that intensive executors are marked as very complex."""
        very_complex = [
            slot
            for slot, complexity in EXECUTOR_COMPLEXITY_MAP.items()
            if complexity == ExecutorComplexity.VERY_COMPLEX
        ]
        assert "D2-Q2" in very_complex
        assert "D3-Q2" in very_complex
        assert "D3-Q3" in very_complex


class TestBatchExecutorBasics:
    """Test basic batch executor functionality."""

    def test_initialization(self, batch_config, mock_method_executor):
        """Test batch executor initialization."""
        executor = BatchExecutor(
            config=batch_config,
            method_executor=mock_method_executor,
        )
        assert executor.config == batch_config
        assert executor.method_executor == mock_method_executor
        assert executor._batch_counter == 0

    def test_get_batch_size_for_executor(self, batch_config, mock_method_executor):
        """Test batch size determination based on complexity."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        batch_size_moderate = executor.get_batch_size_for_executor("D1-Q1")
        assert batch_size_moderate == 20

        batch_size_complex = executor.get_batch_size_for_executor("D3-Q2")
        assert batch_size_complex == 5

        batch_size_unknown = executor.get_batch_size_for_executor("UNKNOWN")
        assert batch_size_unknown == 20

    def test_batch_size_clamping(self, mock_method_executor):
        """Test that batch sizes respect min/max constraints."""
        config = BatchExecutorConfig(
            default_batch_size=10, max_batch_size=15, min_batch_size=8
        )
        executor = BatchExecutor(config=config, method_executor=mock_method_executor)

        batch_size = executor.get_batch_size_for_executor("D1-Q1")
        assert 8 <= batch_size <= 15

    def test_create_batches(self, batch_config, mock_method_executor):
        """Test batch creation from items."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(25))
        batches = executor._create_batches(items, batch_size=10)

        assert len(batches) == 3
        assert batches[0] == (0, list(range(0, 10)))
        assert batches[1] == (1, list(range(10, 20)))
        assert batches[2] == (2, list(range(20, 25)))

    def test_create_batches_exact_fit(self, batch_config, mock_method_executor):
        """Test batch creation with exact fit."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(20))
        batches = executor._create_batches(items, batch_size=10)

        assert len(batches) == 2
        assert all(len(batch[1]) == 10 for batch in batches)

    def test_create_batches_single_batch(self, batch_config, mock_method_executor):
        """Test batch creation with items fitting in single batch."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(5))
        batches = executor._create_batches(items, batch_size=10)

        assert len(batches) == 1
        assert batches[0] == (0, items)


class TestBatchExecution:
    """Test batch execution."""

    @pytest.mark.asyncio
    async def test_execute_single_batch_success(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test successful execution of a single batch."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )
        mock_executor_instance = MockExecutor(
            mock_method_executor, None, batch_config, None, None
        )

        batch_items = [1, 2, 3]
        result = await executor._execute_single_batch(
            "test_batch_001",
            0,
            batch_items,
            mock_executor_instance,
            sample_document,
            sample_question_context,
        )

        assert result.batch_id == "test_batch_001"
        assert result.batch_index == 0
        assert result.status == BatchStatus.COMPLETED
        assert result.metrics["successful_items"] == 3
        assert result.metrics["failed_items"] == 0
        assert len(result.results) == 3
        assert all(r is not None for r in result.results)

    @pytest.mark.asyncio
    async def test_execute_single_batch_all_failures(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test batch execution with all failures."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )
        mock_executor_instance = FailingExecutor(
            mock_method_executor, None, batch_config, None, None
        )

        batch_items = [1, 2, 3]
        result = await executor._execute_single_batch(
            "test_batch_002",
            0,
            batch_items,
            mock_executor_instance,
            sample_document,
            sample_question_context,
        )

        assert result.status == BatchStatus.FAILED
        assert result.metrics["successful_items"] == 0
        assert result.metrics["failed_items"] == 3
        assert len(result.errors) == 3
        assert all(r is None for r in result.results)

    @pytest.mark.asyncio
    async def test_execute_single_batch_partial_success(
        self, mock_method_executor, sample_document, sample_question_context
    ):
        """Test batch execution with partial success."""
        config = BatchExecutorConfig(error_threshold=0.6, enable_instrumentation=False)
        executor = BatchExecutor(config=config, method_executor=mock_method_executor)
        mock_executor_instance = PartialFailExecutor(
            mock_method_executor, None, config, None, None
        )

        batch_items = [1, 2, 3, 4]
        result = await executor._execute_single_batch(
            "test_batch_003",
            0,
            batch_items,
            mock_executor_instance,
            sample_document,
            sample_question_context,
        )

        assert result.status == BatchStatus.PARTIAL_SUCCESS
        assert result.metrics["successful_items"] == 2
        assert result.metrics["failed_items"] == 2
        assert len(result.get_successful_results()) == 2
        assert len(result.get_failed_items()) == 2

    @pytest.mark.asyncio
    async def test_execute_batch_with_retry_success_first_try(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test batch retry logic with success on first try."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )
        mock_executor_instance = MockExecutor(
            mock_method_executor, None, batch_config, None, None
        )

        batch_items = [1, 2]
        result = await executor._execute_batch_with_retry(
            "test_batch_004",
            0,
            batch_items,
            mock_executor_instance,
            sample_document,
            sample_question_context,
        )

        assert result.status == BatchStatus.COMPLETED
        assert result.metrics["retries_used"] == 0

    @pytest.mark.asyncio
    async def test_execute_batch_with_retry_exhausted(
        self, mock_method_executor, sample_document, sample_question_context
    ):
        """Test batch retry logic with retries exhausted."""
        config = BatchExecutorConfig(max_retries=1, enable_instrumentation=False)
        executor = BatchExecutor(config=config, method_executor=mock_method_executor)
        mock_executor_instance = FailingExecutor(
            mock_method_executor, None, config, None, None
        )

        batch_items = [1, 2]
        result = await executor._execute_batch_with_retry(
            "test_batch_005",
            0,
            batch_items,
            mock_executor_instance,
            sample_document,
            sample_question_context,
        )

        assert result.status == BatchStatus.FAILED
        assert result.metrics["retries_used"] == 1


class TestStreamingBatchExecution:
    """Test streaming batch execution."""

    @pytest.mark.asyncio
    async def test_execute_batches_streaming(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test streaming batch execution."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(25))
        batch_results = []

        async for batch_result in executor.execute_batches(
            items, MockExecutor, sample_document, sample_question_context, batch_size=10
        ):
            batch_results.append(batch_result)

        assert len(batch_results) == 3
        assert batch_results[0].batch_index == 0
        assert batch_results[1].batch_index == 1
        assert batch_results[2].batch_index == 2
        assert all(br.status == BatchStatus.COMPLETED for br in batch_results)

    @pytest.mark.asyncio
    async def test_execute_batches_with_base_slot(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test batch execution with base slot for complexity-based sizing."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(30))
        batch_results = []

        async for batch_result in executor.execute_batches(
            items,
            MockExecutor,
            sample_document,
            sample_question_context,
            base_slot="D3-Q2",
        ):
            batch_results.append(batch_result)

        assert len(batch_results) == 6

    @pytest.mark.asyncio
    async def test_execute_batches_empty_items(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test batch execution with empty items list."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        batch_results = []
        async for batch_result in executor.execute_batches(
            [], MockExecutor, sample_document, sample_question_context
        ):
            batch_results.append(batch_result)

        assert len(batch_results) == 0


```

```
class TestParallelBatchExecution:
    """Test parallel batch execution."""

    @pytest.mark.asyncio
    async def test_execute_batches_parallel(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test parallel batch execution."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(40))
        batch_results = await executor.execute_batches_parallel(
            items,
            MockExecutor,
            sample_document,
            sample_question_context,
            batch_size=10,
            max_concurrent_batches=2,
        )

        assert len(batch_results) == 4
        assert all(br.status == BatchStatus.COMPLETED for br in batch_results)
        assert sum(br.metrics["successful_items"] for br in batch_results) == 40

    @pytest.mark.asyncio
    async def test_execute_batches_parallel_with_failures(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test parallel batch execution with some failures."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(20))
        batch_results = await executor.execute_batches_parallel(
            items,
            PartialFailExecutor,
            sample_document,
            sample_question_context,
            batch_size=10,
            max_concurrent_batches=2,
        )

        assert len(batch_results) == 2
        successful_total = sum(br.metrics["successful_items"] for br in batch_results)
        failed_total = sum(br.metrics["failed_items"] for br in batch_results)
        assert successful_total == 10
        assert failed_total == 10


class TestBatchAggregation:
    """Test batch result aggregation."""

    @pytest.mark.asyncio
    async def test_aggregate_batch_results(self, batch_config, mock_method_executor):
        """Test aggregation of batch results."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        batch_results = [
            BatchResult(
                batch_id="batch_001",
                batch_index=0,
                items=[1, 2, 3],
                results=[{"a": 1}, {"a": 2}, {"a": 3}],
                status=BatchStatus.COMPLETED,
                metrics={
                    "batch_id": "batch_001",
                    "batch_index": 0,
                    "batch_size": 3,
                    "status": BatchStatus.COMPLETED,
                    "start_time": 0.0,
                    "end_time": 1.0,
                    "execution_time_ms": 1000.0,
                    "successful_items": 3,
                    "failed_items": 0,
                    "retries_used": 0,
                    "error_messages": [],
                },
            ),
            BatchResult(
                batch_id="batch_002",
                batch_index=1,
                items=[4, 5],
                results=[{"a": 4}, None],
                status=BatchStatus.PARTIAL_SUCCESS,
                metrics={
                    "batch_id": "batch_002",
                    "batch_index": 1,
                    "batch_size": 2,
                    "status": BatchStatus.PARTIAL_SUCCESS,
                    "start_time": 1.0,
                    "end_time": 2.0,
                    "execution_time_ms": 1000.0,
                    "successful_items": 1,
                    "failed_items": 1,
                    "retries_used": 1,
                    "error_messages": ["error"],
                },
                errors=[{"error": "test"}],
            ),
        ]

        aggregated = await executor.aggregate_batch_results(batch_results)

        assert aggregated.total_batches == 2
        assert aggregated.total_items == 5
        assert aggregated.successful_items == 4
        assert aggregated.failed_items == 1
        assert len(aggregated.results) == 4
        assert len(aggregated.errors) == 1
        assert aggregated.success_rate() == 0.8

    @pytest.mark.asyncio
    async def test_aggregate_empty_results(self, batch_config, mock_method_executor):
        """Test aggregation with no results."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        aggregated = await executor.aggregate_batch_results([])

        assert aggregated.total_batches == 0
        assert aggregated.total_items == 0
        assert aggregated.success_rate() == 0.0

    @pytest.mark.asyncio
    async def test_stream_aggregate_batches(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test streaming aggregation of batch results."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(15))

        def accumulator_fn(acc, result):
            """Accumulate call counts."""
            call_count = result.get("call_count", 0)
            if "total_calls" not in acc:
                acc["total_calls"] = 0
            acc["total_calls"] += call_count
            return acc

        batch_stream = executor.execute_batches(
            items, MockExecutor, sample_document, sample_question_context, batch_size=5
        )

        final = await executor.stream_aggregate_batches(batch_stream, accumulator_fn)

        assert final is not None
        assert "total_calls" in final
```

```python


class TestBatchResultHelpers:
    """Test BatchResult helper methods."""

    def test_is_successful(self):
        """Test is_successful helper."""
        result = BatchResult(
            batch_id="test",
            batch_index=0,
            items=[1, 2],
            results=[{"a": 1}, {"a": 2}],
            status=BatchStatus.COMPLETED,
            metrics={
                "batch_id": "test",
                "batch_index": 0,
                "batch_size": 2,
                "status": BatchStatus.COMPLETED,
                "start_time": 0.0,
                "end_time": 1.0,
                "execution_time_ms": 1000.0,
                "successful_items": 2,
                "failed_items": 0,
                "retries_used": 0,
                "error_messages": [],
            },
        )
        assert result.is_successful() is True

    def test_has_partial_success(self):
        """Test has_partial_success helper."""
        result = BatchResult(
            batch_id="test",
            batch_index=0,
            items=[1, 2],
            results=[{"a": 1}, None],
            status=BatchStatus.PARTIAL_SUCCESS,
            metrics={
                "batch_id": "test",
                "batch_index": 0,
                "batch_size": 2,
                "status": BatchStatus.PARTIAL_SUCCESS,
                "start_time": 0.0,
                "end_time": 1.0,
                "execution_time_ms": 1000.0,
                "successful_items": 1,
                "failed_items": 1,
                "retries_used": 0,
                "error_messages": ["error"],
            },
        )
        assert result.has_partial_success() is True

    def test_get_successful_results(self):
        """Test get_successful_results helper."""
        result = BatchResult(
            batch_id="test",
            batch_index=0,
            items=[1, 2, 3],
            results=[{"a": 1}, None, {"a": 3}],
            status=BatchStatus.PARTIAL_SUCCESS,
            metrics={
                "batch_id": "test",
                "batch_index": 0,
                "batch_size": 3,
                "status": BatchStatus.PARTIAL_SUCCESS,
                "start_time": 0.0,
                "end_time": 1.0,
                "execution_time_ms": 1000.0,
                "successful_items": 2,
                "failed_items": 1,
                "retries_used": 0,
                "error_messages": ["error"],
            },
        )
        successful = result.get_successful_results()
        assert len(successful) == 2
        assert successful[0] == (1, {"a": 1})
        assert successful[1] == (3, {"a": 3})

    def test_get_failed_items(self):
        """Test get_failed_items helper."""
        result = BatchResult(
            batch_id="test",
            batch_index=0,
            items=[1, 2, 3],
            results=[{"a": 1}, None, {"a": 3}],
            status=BatchStatus.PARTIAL_SUCCESS,
            metrics={
                "batch_id": "test",
                "batch_index": 0,
                "batch_size": 3,
                "status": BatchStatus.PARTIAL_SUCCESS,
                "start_time": 0.0,
                "end_time": 1.0,
                "execution_time_ms": 1000.0,
                "successful_items": 2,
                "failed_items": 1,
                "retries_used": 0,
                "error_messages": ["error"],
            },
        )
        failed = result.get_failed_items()
        assert len(failed) == 1
        assert failed[0] == 2


class TestEdgeCases:
    """Test edge cases and error conditions."""

    @pytest.mark.asyncio
    async def test_execute_batches_single_item(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test batch execution with single item."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = [1]
        batch_results = []

        async for batch_result in executor.execute_batches(
            items, MockExecutor, sample_document, sample_question_context, batch_size=10
        ):
            batch_results.append(batch_result)

        assert len(batch_results) == 1
        assert batch_results[0].metrics["batch_size"] == 1

    @pytest.mark.asyncio
    async def test_batch_counter_increments(
        self,
        batch_config,
        mock_method_executor,
        sample_document,
        sample_question_context,
    ):
        """Test that batch counter increments properly."""
        executor = BatchExecutor(
            config=batch_config, method_executor=mock_method_executor
        )

        items = list(range(15))
        batch_results = []

        async for batch_result in executor.execute_batches(
            items, MockExecutor, sample_document, sample_question_context, batch_size=5
        ):
            batch_results.append(batch_result)

        assert executor._batch_counter == 3
        assert batch_results[0].batch_id == "batch_000001"
        assert batch_results[1].batch_id == "batch_000002"
        assert batch_results[2].batch_id == "batch_000003"

    def test_aggregated_results_success_rate_zero_items(self):
        """Test success rate calculation with zero items."""
        aggregated = AggregatedBatchResults(
            total_batches=0,
            total_items=0,
            successful_items=0,
            failed_items=0,
            results=[],
            errors=[],
            execution_time_ms=0.0,
            batch_metrics=[],
        )
        assert aggregated.success_rate() == 0.0



================================================================================
FILE: tests/test_chunk_semantic_auditor.py
================================================================================

import json
import tempfile
from pathlib import Path

import pytest

from tools.chunk_semantic_auditor import (
    ChunkMetadata,
    ChunkSemanticAuditor,
    SemanticAuditResult,
)


@pytest.fixture
def temp_artifacts_dir():
    with tempfile.TemporaryDirectory() as tmpdir:
        yield Path(tmpdir)


@pytest.fixture
def sample_chunks_file(temp_artifacts_dir):
    chunks_data = {
        "chunks": [
            {
                "chunk_id": "chunk_001",
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
                "text": "The budget allocation for economic development includes funding for small business support programs, entrepreneurship training, and innovation infrastructure with a total investment of $5 million.",
            },
            {
                "chunk_id": "chunk_002",
                "policy_area_id": "PA02",
                "dimension_id": "DIM03",
                "text": "The health program will deliver 50 community clinics, vaccinate 100,000 children, and provide free medical consultations to vulnerable populations.",
            },
            {
                "chunk_id": "chunk_003",
                "policy_area_id": "PA04",
                "dimension_id": "DIM06",
                "text": "The causal link between reforestation activities and improved air quality will be measured through PM2.5 monitoring stations and longitudinal health studies.",
            },
        ]
    }

    chunks_file = temp_artifacts_dir / "test_chunks.json"
    with open(chunks_file, "w") as f:
        json.dump(chunks_data, f)

    return chunks_file


@pytest.fixture
def mismatched_chunks_file(temp_artifacts_dir):
    chunks_data = {
        "chunks": [
            {
                "chunk_id": "chunk_mismatch",
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
                "text": "This text discusses cultural heritage preservation and traditional music festivals, which has nothing to do with economic development or inputs.",
            }
        ]
    }

    chunks_file = temp_artifacts_dir / "mismatched_chunks.json"
    with open(chunks_file, "w") as f:
        json.dump(chunks_data, f)

    return chunks_file


def test_chunk_metadata_creation():
    chunk = ChunkMetadata(
        chunk_id="test_001",
        file_path="test.json",
        policy_area_id="PA01",
        dimension_id="DIM01",
        text_content="Sample text",
    )

    assert chunk.chunk_id == "test_001"
    assert chunk.policy_area_id == "PA01"
    assert chunk.dimension_id == "DIM01"


def test_auditor_initialization(temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.7, verbose=False
    )

    assert auditor.artifacts_dir == temp_artifacts_dir
    assert auditor.threshold == 0.7
    assert auditor.model is None
    assert len(auditor.chunks) == 0

```

```
def test_load_model(temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    auditor.load_model()

    assert auditor.model is not None


def test_discover_chunk_artifacts(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    chunk_files = auditor.discover_chunk_artifacts()

    assert len(chunk_files) >= 1
    assert sample_chunks_file in chunk_files


def test_load_chunk_metadata(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    chunks = auditor.load_chunk_metadata(sample_chunks_file)

    assert len(chunks) == 3
    assert chunks[0].chunk_id == "chunk_001"
    assert chunks[0].policy_area_id == "PA01"
    assert chunks[0].dimension_id == "DIM01"
    assert "economic development" in chunks[0].text_content.lower()


def test_load_all_chunks(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    auditor.load_all_chunks()

    assert len(auditor.chunks) == 3


def test_compute_semantic_coherence(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    auditor.load_model()
    auditor.load_all_chunks()

    chunk = auditor.chunks[0]
    score = auditor.compute_semantic_coherence(chunk)

    assert isinstance(score, float)
    assert 0.0 <= score <= 1.0


def test_audit_chunk_pass(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
    )
    auditor.load_model()
    auditor.load_all_chunks()

    chunk = auditor.chunks[0]
    result = auditor.audit_chunk(chunk)

    assert isinstance(result, SemanticAuditResult)
    assert result.chunk_id == chunk.chunk_id
    assert result.passed is True


def test_audit_chunk_fail(mismatched_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.7, verbose=False
    )
    auditor.load_model()
    auditor.load_all_chunks()

    chunk = auditor.chunks[0]
    result = auditor.audit_chunk(chunk)

    assert isinstance(result, SemanticAuditResult)
    assert result.passed is False
    assert result.coherence_score < 0.7


def test_audit_all_chunks(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
    )
    auditor.load_model()
    auditor.load_all_chunks()
    auditor.audit_all_chunks()

    assert len(auditor.audit_results) == 3
    assert all(isinstance(r, SemanticAuditResult) for r in auditor.audit_results)


def test_generate_report(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
    )
    auditor.load_model()
    auditor.load_all_chunks()
    auditor.audit_all_chunks()

    report = auditor.generate_report()

    assert "metadata" in report
    assert "summary" in report
    assert "failures" in report
    assert report["metadata"]["total_chunks_audited"] == 3
    assert report["summary"]["passed"] >= 0
    assert report["summary"]["failed"] >= 0
    assert report["summary"]["pass_rate"] >= 0.0


def test_save_report(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
    )
    auditor.load_model()
    auditor.load_all_chunks()
    auditor.audit_all_chunks()

    report = auditor.generate_report()
    output_file = temp_artifacts_dir / "test_report.json"
    auditor.save_report(report, output_file)

    assert output_file.exists()

    with open(output_file) as f:
        loaded_report = json.load(f)

    assert loaded_report == report


def test_full_audit_workflow(sample_chunks_file, temp_artifacts_dir):
    auditor = ChunkSemanticAuditor(
        artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
    )

    exit_code = auditor.run()

    assert exit_code in [0, 1]
    assert (temp_artifacts_dir / "semantic_audit_report.json").exists()


def test_empty_artifacts_dir():
    with tempfile.TemporaryDirectory() as tmpdir:
        auditor = ChunkSemanticAuditor(artifacts_dir=Path(tmpdir), verbose=False)
        auditor.load_model()
        auditor.load_all_chunks()

        assert len(auditor.chunks) == 0


def test_invalid_json_file(temp_artifacts_dir):
    invalid_file = temp_artifacts_dir / "invalid_chunks.json"
    with open(invalid_file, "w") as f:
        f.write("{ invalid json }")

    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    chunks = auditor.load_chunk_metadata(invalid_file)

    assert len(chunks) == 0


def test_missing_metadata_fields(temp_artifacts_dir):
    incomplete_chunks = {
        "chunks": [
            {"chunk_id": "c1", "text": "text only"},
            {"policy_area_id": "PA01", "text": "missing dimension"},
            {"dimension_id": "DIM01", "text": "missing policy area"},
        ]
    }

    chunks_file = temp_artifacts_dir / "incomplete.json"
    with open(chunks_file, "w") as f:
        json.dump(incomplete_chunks, f)

    auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
    chunks = auditor.load_chunk_metadata(chunks_file)

    assert len(chunks) == 0



================================================================================
FILE: tests/test_config_manager.py
================================================================================

"""Tests for configuration management system."""

import json
import shutil
import tempfile
import time
from datetime import datetime
from pathlib import Path

import pytest

from system.config.config_manager import ConfigManager


@pytest.fixture
def temp_config_dir():
    """Create temporary config directory for testing."""
    temp_dir = tempfile.mkdtemp()
    yield Path(temp_dir)
    shutil.rmtree(temp_dir)


@pytest.fixture
def config_manager(temp_config_dir):
    """Create ConfigManager instance for testing."""
    return ConfigManager(config_root=temp_config_dir)


class TestConfigManager:
    """Test suite for ConfigManager."""

    def test_init_creates_directories(self, temp_config_dir):
        """Test that initialization creates required directories."""
        ConfigManager(config_root=temp_config_dir)

        assert (temp_config_dir / ".backup").exists()
        assert (temp_config_dir / "calibration").exists()
        assert (temp_config_dir / "questionnaire").exists()
        assert (temp_config_dir / "environments").exists()

    def test_save_config_creates_file(self, config_manager, temp_config_dir):
        """Test saving configuration file."""
        content = "test content"
        file_path = config_manager.save_config("test/config.txt", content)

        assert file_path.exists()
        assert file_path.read_text() == content

    def test_save_config_json(self, config_manager, temp_config_dir):
        """Test saving JSON configuration."""
        data = {"key": "value", "number": 42, "nested": {"a": 1}}
        file_path = config_manager.save_config_json("test/config.json", data)

        assert file_path.exists()
        loaded_data = json.loads(file_path.read_text())
        assert loaded_data == data

    def test_save_updates_registry(self, config_manager, temp_config_dir):
        """Test that saving a file updates the hash registry."""
        config_manager.save_config("test/file.txt", "content")

        registry = config_manager.get_registry()
        assert "test/file.txt" in registry
        assert "hash" in registry["test/file.txt"]
        assert "last_modified" in registry["test/file.txt"]
        assert "size_bytes" in registry["test/file.txt"]

    def test_hash_computation_deterministic(self, config_manager, temp_config_dir):
        """Test that SHA256 hash computation is deterministic."""
        content = "test content for hashing"
        config_manager.save_config("test/hash.txt", content)

        info1 = config_manager.get_file_info("test/hash.txt")
        config_manager.save_config("test/hash.txt", content, create_backup=False)
        info2 = config_manager.get_file_info("test/hash.txt")

        assert info1["hash"] == info2["hash"]

    def test_hash_changes_on_modification(self, config_manager, temp_config_dir):
        """Test that hash changes when file is modified."""
        config_manager.save_config("test/file.txt", "original content")
        original_hash = config_manager.get_file_info("test/file.txt")["hash"]

        config_manager.save_config("test/file.txt", "modified content")
        modified_hash = config_manager.get_file_info("test/file.txt")["hash"]

        assert original_hash != modified_hash

    def test_backup_created_on_save(self, config_manager, temp_config_dir):
        """Test that backup is created before modification."""
        config_manager.save_config("test/file.txt", "original")
        time.sleep(0.001)
        config_manager.save_config("test/file.txt", "modified")

        backups = config_manager.list_backups("test/file.txt")
        assert len(backups) >= 1

        backup_content = backups[0].read_text()
        assert backup_content == "original"

    def test_backup_filename_format(self, config_manager, temp_config_dir):
        """Test that backup filenames follow YYYYMMDD_HHMMSS_microseconds format."""
        config_manager.save_config("calibration/model.json", '{"test": true}')
        time.sleep(0.001)
        config_manager.save_config("calibration/model.json", '{"test": false}')

        backups = config_manager.list_backups("calibration/model.json")
        assert len(backups) >= 1

        backup_name = backups[0].name
        parts = backup_name.split("_")
        timestamp_part = parts[0] + "_" + parts[1]

        try:
            datetime.strptime(timestamp_part, "%Y%m%d_%H%M%S")
        except ValueError:
            pytest.fail(
                f"Backup filename doesn't match YYYYMMDD_HHMMSS_microseconds format: {backup_name}"
            )

    def test_no_backup_on_first_save(self, config_manager, temp_config_dir):
        """Test that no backup is created for new files."""
        config_manager.save_config("test/new.txt", "first save")

        backups = config_manager.list_backups("test/new.txt")
        assert len(backups) == 0

    def test_load_config(self, config_manager, temp_config_dir):
        """Test loading configuration file."""
        content = "test content"
        config_manager.save_config("test/file.txt", content)

        loaded_content = config_manager.load_config("test/file.txt")
        assert loaded_content == content

    def test_load_config_json(self, config_manager, temp_config_dir):
        """Test loading JSON configuration."""
        data = {"key": "value", "list": [1, 2, 3]}
        config_manager.save_config_json("test/data.json", data)

        loaded_data = config_manager.load_config_json("test/data.json")
        assert loaded_data == data

    def test_verify_hash_valid(self, config_manager, temp_config_dir):
        """Test hash verification for unchanged file."""
        config_manager.save_config("test/file.txt", "content")
        assert config_manager.verify_hash("test/file.txt") is True

    def test_verify_hash_invalid_after_external_modification(
        self, config_manager, temp_config_dir
    ):
        """Test hash verification fails after external modification."""
        file_path = config_manager.save_config("test/file.txt", "original")

        file_path.write_text("externally modified")
        assert config_manager.verify_hash("test/file.txt") is False

    def test_verify_hash_nonexistent_file(self, config_manager, temp_config_dir):
        """Test hash verification for nonexistent file."""
        assert config_manager.verify_hash("nonexistent.txt") is False

    def test_list_backups_sorted_by_timestamp(self, config_manager, temp_config_dir):
        """Test that backups are sorted with newest first."""
        config_manager.save_config("test/file.txt", "version 1")
        time.sleep(0.001)
        config_manager.save_config("test/file.txt", "version 2")
        time.sleep(0.001)
        config_manager.save_config("test/file.txt", "version 3")

        backups = config_manager.list_backups("test/file.txt")
        assert len(backups) >= 2

        for i in range(len(backups) - 1):
            assert backups[i].name > backups[i + 1].name

    def test_list_backups_all(self, config_manager, temp_config_dir):
        """Test listing all backups without filter."""
        config_manager.save_config("file1.txt", "content")
        config_manager.save_config("file1.txt", "modified")

        config_manager.save_config("file2.txt", "content")
        config_manager.save_config("file2.txt", "modified")

        all_backups = config_manager.list_backups()
        assert len(all_backups) >= 2

    def test_restore_backup(self, config_manager, temp_config_dir):
        """Test restoring from backup."""
        original_content = "original version"
        config_manager.save_config("test/file.txt", original_content)
        time.sleep(0.001)

        config_manager.save_config("test/file.txt", "modified version")

        backups = config_manager.list_backups("test/file.txt")
        config_manager.restore_backup(backups[0])

        restored_content = config_manager.load_config("test/file.txt")
        assert restored_content == original_content

    def test_restore_creates_backup(self, config_manager, temp_config_dir):
        """Test that restore creates backup of current version."""
        config_manager.save_config("test/file.txt", "version 1")
        time.sleep(0.001)
        config_manager.save_config("test/file.txt", "version 2")

        backups_before = len(config_manager.list_backups("test/file.txt"))

        first_backup = config_manager.list_backups("test/file.txt")[-1]
        config_manager.restore_backup(first_backup)
```

```

            backups_after = len(config_manager.list_backups("test/file.txt"))
            assert backups_after == backups_before + 1

        def test_rebuild_registry(self, config_manager, temp_config_dir):
            """Test rebuilding hash registry from disk."""
            config_manager.save_config("file1.json", '{"a": 1}')
            config_manager.save_config("file2.yaml", "key: value")

            registry_file = temp_config_dir / "config_hash_registry.json"
            registry_file.unlink()

            rebuilt_registry = config_manager.rebuild_registry()

            assert "file1.json" in rebuilt_registry
            assert "file2.yaml" in rebuilt_registry

        def test_get_file_info_nonexistent(self, config_manager, temp_config_dir):
            """Test getting info for nonexistent file."""
            info = config_manager.get_file_info("nonexistent.txt")
            assert info is None

        def test_registry_persistence(self, temp_config_dir):
            """Test that registry persists across manager instances."""
            manager1 = ConfigManager(config_root=temp_config_dir)
            manager1.save_config("test/file.txt", "content")
            original_hash = manager1.get_file_info("test/file.txt")["hash"]

            manager2 = ConfigManager(config_root=temp_config_dir)
            loaded_hash = manager2.get_file_info("test/file.txt")["hash"]

            assert original_hash == loaded_hash

        def test_save_without_backup(self, config_manager, temp_config_dir):
            """Test saving without creating backup."""
            config_manager.save_config("test/file.txt", "original")
            config_manager.save_config("test/file.txt", "modified", create_backup=False)

            backups = config_manager.list_backups("test/file.txt")
            assert len(backups) == 0

        def test_nested_directory_creation(self, config_manager, temp_config_dir):
            """Test that nested directories are created automatically."""
            config_manager.save_config("a/b/c/deep.json", '{"nested": true}')

            file_path = temp_config_dir / "a/b/c/deep.json"
            assert file_path.exists()

        def test_json_sorted_keys(self, config_manager, temp_config_dir):
            """Test that JSON is saved with sorted keys."""
            data = {"z": 1, "a": 2, "m": 3}
            file_path = config_manager.save_config_json("test/sorted.json", data)

            content = file_path.read_text()
            assert content.index('"a"') < content.index('"m"') < content.index('"z"')

```

```
    def test_unicode_content_handling(self, config_manager, temp_config_dir):
        """Test handling of Unicode content."""
        content = "Hello ä¸\226ç\225\214 ð\237\214\215"
        config_manager.save_config("test/unicode.txt", content)

        loaded = config_manager.load_config("test/unicode.txt")
        assert loaded == content



================================================================================
FILE: tests/test_intrinsic_pipeline_behavior.py
================================================================================

"""
test_intrinsic_pipeline_behavior.py - Test pipeline execution with intrinsic calibration

Tests fallback behavior:
1. All computed: Normal execution with actual @b values
2. Mix of pending/excluded: Verify fallback behavior (pendingâ\206\222@b=0.5, excludedâ\206\222skip, noneâ\206\222@b=0.3 with warning)

DEPRECATED: This test relies on outdated IntrinsicCalibrationLoader interface.
See tests/DEPRECATED_TESTS.md for details.
"""
import json
import tempfile
from pathlib import Path

import pytest

from src.farfan_pipeline.core.calibration.intrinsic_calibration_loader import (
    IntrinsicCalibrationLoader,
)

pytestmark = pytest.mark.obsolete


@pytest.fixture
def temp_calibration_file():
    """Create a temporary calibration file for testing."""
    data = {
        "_metadata": {
            "version": "1.0.0",
            "generated": "2025-01-15T00:00:00Z",
            "description": "Test calibration data",
            "total_methods": 5,
            "computed_methods": 4,
            "coverage_percent": 80.0
        },
        "TestClass.computed_method": {
            "intrinsic_score": [0.75, 0.85],
            "b_theory": 0.80,
            "b_impl": 0.78,
            "b_deploy": 0.83,
            "calibration_status": "computed",
            "layer": "engine",
            "last_updated": "2025-01-15T00:00:00Z"
        },
        "TestClass.pending_method": {
            "intrinsic_score": [0.0, 0.0],
            "b_theory": 0.0,
            "b_impl": 0.0,
            "b_deploy": 0.0,
            "calibration_status": "pending",
            "layer": "processor",
            "last_updated": "2025-01-15T00:00:00Z"
        },
        "TestClass.excluded_method": {
            "intrinsic_score": [0.0, 0.0],
            "b_theory": 0.0,
            "b_impl": 0.0,
            "b_deploy": 0.0,
            "calibration_status": "excluded",
            "layer": "utility",
            "last_updated": "2025-01-15T00:00:00Z"
        },
        "TestClass.none_method": {
            "intrinsic_score": [0.0, 0.0],
            "b_theory": 0.0,
            "b_impl": 0.0,
            "b_deploy": 0.0,
            "calibration_status": "none",
            "layer": "engine",
            "last_updated": "2025-01-15T00:00:00Z"
        },
        "TestClass.another_computed": {
            "intrinsic_score": [0.68, 0.78],
            "b_theory": 0.73,
            "b_impl": 0.70,
            "b_deploy": 0.76,
            "calibration_status": "computed",
            "layer": "processor",
            "last_updated": "2025-01-15T00:00:00Z"
        }
    }

    with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:
        json.dump(data, f, indent=2)
        temp_path = f.name

    yield temp_path
    Path(temp_path).unlink()


def test_loader_initialization(temp_calibration_file):
    """Test loader initializes correctly with valid data."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    metadata = loader.get_metadata()

    assert metadata["coverage_percent"] >= 80.0
    assert metadata["computed_methods"] == 4

```

```

def test_loader_rejects_low_coverage():
    """Test loader rejects data with <80% coverage."""
    data = {
        "_metadata": {
            "version": "1.0.0",
            "generated": "2025-01-15T00:00:00Z",
            "description": "Low coverage test",
            "total_methods": 100,
            "computed_methods": 70,
            "coverage_percent": 70.0
        }
    }

    with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:
        json.dump(data, f, indent=2)
        temp_path = f.name

    try:
        with pytest.raises(ValueError, match="coverage.*80"):
            IntrinsicCalibrationLoader(temp_path)
    finally:
        Path(temp_path).unlink()


def test_computed_method_returns_actual_values(temp_calibration_file):
    """Test computed method returns actual @b values from JSON."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.computed_method")

    assert cal is not None
    assert cal.calibration_status == "computed"
    assert cal.b_theory == 0.80
    assert cal.b_impl == 0.78
    assert cal.b_deploy == 0.83
    assert cal.intrinsic_score == (0.75, 0.85)
    assert cal.layer == "engine"


def test_pending_method_returns_fallback_05(temp_calibration_file):
    """Test pending method returns @b=0.5 fallback."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.pending_method")

    assert cal is not None
    assert cal.calibration_status == "pending"
    assert cal.b_theory == 0.5
    assert cal.b_impl == 0.5
    assert cal.b_deploy == 0.5
    assert cal.intrinsic_score == (0.48, 0.52)


def test_excluded_method_returns_none(temp_calibration_file):
    """Test excluded method returns None (signals skip)."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.excluded_method")

    assert cal is None


def test_none_method_returns_fallback_03(temp_calibration_file):
    """Test none status method returns @b=0.3 fallback."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.none_method")

    assert cal is not None
    assert cal.calibration_status == "none"
    assert cal.b_theory == 0.3
    assert cal.b_impl == 0.3
    assert cal.b_deploy == 0.3
    assert cal.intrinsic_score == (0.28, 0.32)


def test_missing_method_returns_fallback_03(temp_calibration_file):
    """Test method not in registry returns @b=0.3 fallback."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.unknown_method")

    assert cal is not None
    assert cal.calibration_status == "none"
    assert cal.b_theory == 0.3
    assert cal.b_impl == 0.3
    assert cal.b_deploy == 0.3


def test_contamination_detection():
    """Test loader detects contaminated data."""
    data = {
        "_metadata": {
            "version": "1.0.0",
            "generated": "2025-01-15T00:00:00Z",
            "description": "Contaminated test",
            "total_methods": 1,
            "computed_methods": 1,
            "coverage_percent": 100.0
        },
        "TestClass.contaminated_method": {
            "intrinsic_score": [0.75, 0.85],
            "b_theory": 0.80,
            "b_impl": 0.78,
            "b_deploy": 0.83,
            "final_score": 0.85,  # CONTAMINATION
            "calibration_status": "computed",
            "layer": "engine",
            "last_updated": "2025-01-15T00:00:00Z"
        }
    }

    with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:
        json.dump(data, f, indent=2)
        temp_path = f.name

    try:
        loader = IntrinsicCalibrationLoader(temp_path)
        with pytest.raises(ValueError, match="CONTAMINATION"):
            loader.verify_purity()
    finally:
        Path(temp_path).unlink()


def test_composite_b_calculation(temp_calibration_file):
    """Test composite @b score calculation."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)
    cal = loader.get_calibration("TestClass.computed_method")

    expected_composite = (0.80 + 0.78 + 0.83) / 3.0
    assert abs(cal.get_composite_b() - expected_composite) < 0.001


def test_pipeline_execution_all_computed(temp_calibration_file):
    """Simulate pipeline execution with all computed methods."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)

    methods_to_execute = [
        "TestClass.computed_method",
        "TestClass.another_computed"
    ]

    results = {}
    for method_id in methods_to_execute:
        cal = loader.get_calibration(method_id)
        if cal is not None:  # Not excluded
            results[method_id] = {
                "executed": True,
                "composite_b": cal.get_composite_b(),
                "status": cal.calibration_status
            }

    assert len(results) == 2
    assert all(r["executed"] for r in results.values())
    assert all(r["status"] == "computed" for r in results.values())


def test_pipeline_execution_mixed_statuses(temp_calibration_file):
    """Simulate pipeline execution with mixed calibration statuses."""
    loader = IntrinsicCalibrationLoader(temp_calibration_file)

    methods_to_execute = [
        "TestClass.computed_method",
        "TestClass.pending_method",
        "TestClass.excluded_method",
        "TestClass.none_method"
    ]

    results = {}
    skipped = []

    for method_id in methods_to_execute:
        cal = loader.get_calibration(method_id)
        if cal is None:  # Excluded
            skipped.append(method_id)
        else:
            results[method_id] = {
                "executed": True,
                "composite_b": cal.get_composite_b(),
                "status": cal.calibration_status,
                "is_fallback": cal.calibration_status != "computed"
            }

    assert len(results) == 3  # computed, pending, none
    assert len(skipped) == 1  # excluded
    assert "TestClass.excluded_method" in skipped

    # Verify fallback behavior
    assert results["TestClass.pending_method"]["composite_b"] == 0.5
    assert results["TestClass.none_method"]["composite_b"] == 0.3
    assert results["TestClass.computed_method"]["composite_b"] > 0.7


def test_real_calibration_file_exists():
    """Test that the real calibration file exists and is valid."""
    cal_path = Path("config/intrinsic_calibration.json")
    assert cal_path.exists(), "Real calibration file must exist"

    loader = IntrinsicCalibrationLoader(str(cal_path))
    metadata = loader.get_metadata()

    assert metadata["coverage_percent"] >= 80.0
    assert metadata["computed_methods"] >= 30

    # Verify purity
    assert loader.verify_purity()


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



================================================================================
FILE: tests/test_intrinsic_purity.py
================================================================================

"""
test_intrinsic_purity.py - Verification of @b-only enforcement in intrinsic calibration

This test ensures strict @b-only enforcement:
- NO keys matching @chain, @q, @d, @p, @C, @u, @m, final_score, layer_scores
- ONLY keys: intrinsic_score, b_theory, b_impl, b_deploy, calibration_status, layer, last_updated
- Coverage >= 80% (computed methods / total methods)
- At least 30 executors with status='computed'
"""
import json
from pathlib import Path
```

```

import pytest


def test_intrinsic_calibration_exists():
    """Verify intrinsic_calibration.json exists."""
    cal_path = Path("config/intrinsic_calibration.json")
    assert cal_path.exists(), "config/intrinsic_calibration.json must exist"


def test_no_contaminated_keys():
    """CRITICAL: Fail if ANY non-@b layer data exists in JSON."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    forbidden_patterns = ["@chain", "@q", "@d", "@p", "@C", "@u", "@m",
                          "final_score", "layer_scores", "chain_", "queue_",
                          "downstream_", "upstream_", "composite_"]

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        for key in method_data:
            for pattern in forbidden_patterns:
                assert pattern not in key.lower(), (
                    f"CONTAMINATION DETECTED: method '{method_id}' contains "
                    f"forbidden key '{key}' matching pattern '{pattern}'. "
                    f"Intrinsic calibration incomplete or contaminated."
                )


def test_only_b_layer_keys():
    """Verify each method entry contains ONLY @b-layer keys."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    allowed_keys = {"intrinsic_score", "b_theory", "b_impl", "b_deploy",
                    "calibration_status", "layer", "last_updated"}

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        actual_keys = set(method_data.keys())
        extra_keys = actual_keys - allowed_keys
        missing_keys = allowed_keys - actual_keys

        assert not extra_keys, (
            f"Method '{method_id}' has unauthorized keys: {extra_keys}. "
            f"Only {allowed_keys} are allowed."
        )
        assert not missing_keys, (
            f"Method '{method_id}' is missing required keys: {missing_keys}"
        )


def test_coverage_requirement():
    """Verify >=80% coverage (computed methods / total methods)."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    metadata = data["_metadata"]
    total = metadata["total_methods"]
    computed = metadata["computed_methods"]
    coverage = metadata["coverage_percent"]

    assert coverage >= 80.0, (
        f"Coverage requirement FAILED: {coverage}% < 80%. "
        f"Need at least {int(total * 0.8)} computed methods, have {computed}. "
        f"Intrinsic calibration incomplete or contaminated."
    )


def test_minimum_computed_executors():
    """Verify at least 30 executors have status='computed'."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    computed_count = sum(
        1 for method_id, method_data in data.items()
        if method_id != "_metadata" and method_data.get("calibration_status") == "computed"
    )

    assert computed_count >= 30, (
        f"Minimum executor requirement FAILED: {computed_count} < 30 computed methods. "
        f"Intrinsic calibration incomplete or contaminated."
    )


def test_valid_intrinsic_scores():
    """Verify intrinsic_score is [low, high] with valid values."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        if method_data.get("calibration_status") == "computed":
            score = method_data.get("intrinsic_score")
            assert isinstance(score, list) and len(score) == 2, (
                f"Method '{method_id}' has invalid intrinsic_score format: {score}"
            )
            assert 0.0 <= score[0] <= 1.0 and 0.0 <= score[1] <= 1.0, (
                f"Method '{method_id}' has intrinsic_score out of range [0,1]: {score}"
            )
            assert score[0] <= score[1], (
                f"Method '{method_id}' has invalid intrinsic_score (low > high): {score}"
            )


def test_valid_b_scores():
    """Verify b_theory, b_impl, b_deploy are valid floats in [0,1]."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        if method_data.get("calibration_status") == "computed":
            for b_key in ["b_theory", "b_impl", "b_deploy"]:
                b_val = method_data.get(b_key)
                assert isinstance(b_val, int | float), (
                    f"Method '{method_id}' has invalid {b_key} type: {type(b_val)}"
                )
                assert 0.0 <= b_val <= 1.0, (
                    f"Method '{method_id}' has {b_key} out of range [0,1]: {b_val}"
                )


def test_calibration_status_values():
    """Verify calibration_status is one of: computed, pending, excluded, none."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    valid_statuses = {"computed", "pending", "excluded", "none"}

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        status = method_data.get("calibration_status")
        assert status in valid_statuses, (
            f"Method '{method_id}' has invalid calibration_status: '{status}'. "
            f"Must be one of {valid_statuses}"
        )


def test_layer_values():
    """Verify layer is one of: engine, processor, utility."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    valid_layers = {"engine", "processor", "utility"}

    for method_id, method_data in data.items():
        if method_id == "_metadata":
            continue

        layer = method_data.get("layer")
        assert layer in valid_layers, (
            f"Method '{method_id}' has invalid layer: '{layer}'. "
            f"Must be one of {valid_layers}"
        )


def test_metadata_structure():
    """Verify _metadata contains required fields."""
    cal_path = Path("config/intrinsic_calibration.json")
    with open(cal_path) as f:
        data = json.load(f)

    assert "_metadata" in data, "JSON must contain _metadata"
    metadata = data["_metadata"]

    required_fields = {"version", "generated", "description", "total_methods",
                       "computed_methods", "coverage_percent"}
    actual_fields = set(metadata.keys())

    missing = required_fields - actual_fields
    assert not missing, f"_metadata missing required fields: {missing}"


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



================================================================================
FILE: tests/test_inventory_completeness.py
================================================================================

#!/usr/bin/env python3
"""Test inventory completeness - verifies all critical methods are present

Validates the methods_inventory_raw.json file generated by scan_methods_inventory.py.
This test suite ensures proper scanner operation and inventory completeness.

See METHODS_INVENTORY_README.md for details on the scanner workflow and integration.
"""

import json
from pathlib import Path

import pytest

# Test constants
MINIMUM_METHOD_COUNT = 200
MIN_CANONICAL_ID_PARTS = 2
MIN_CLASS_METHOD_PARTS = 3
MIN_TAG_RATIO = 0.3
MIN_DEREK_BEACH_METHODS = 10
MIN_EXECUTOR_METHODS = 5

```

```
CRITICAL_METHODS = [
    "analysis.derek_beach.CDAFException._format_message",
    "analysis.derek_beach.CDAFException.to_dict",
    "analysis.derek_beach.ConfigLoader._load_config",
    "analysis.derek_beach.ConfigLoader._validate_config",
    "analysis.derek_beach.BeachEvidentialTest.classify_test",
    "analysis.derek_beach.BeachEvidentialTest.apply_test_logic",
    "core.aggregation.AreaPolicyAggregator",
    "core.aggregation.ClusterAggregator",
    "core.aggregation.DimensionAggregator",
    "core.aggregation.MacroAggregator",
    "processing.aggregation.AreaPolicyAggregator",
    "processing.aggregation.ClusterAggregator",
    "processing.aggregation.DimensionAggregator",
    "processing.aggregation.MacroAggregator",
    "core.orchestrator.executors",
    "core.orchestrator.executors_contract",
]


@pytest.fixture
def inventory():
    """Load the inventory JSON file"""
    inventory_path = Path("methods_inventory_raw.json")

    if not inventory_path.exists():
        pytest.fail(f"Inventory file not found: {inventory_path}")

    with open(inventory_path, encoding="utf-8") as f:
        return json.load(f)


def test_inventory_exists():
    """Verify inventory file exists"""
    assert Path("methods_inventory_raw.json").exists(), "Inventory file must exist"


def test_minimum_method_count(inventory):
    """Verify at least 200 methods in inventory"""
    total = inventory["metadata"]["total_methods"]
    assert (
        total >= MINIMUM_METHOD_COUNT
    ), f"Insufficient methods: {total} < {MINIMUM_METHOD_COUNT}"


def test_critical_files_present(inventory):
    """Verify methods from critical files are present"""
    methods = inventory["methods"]
    source_files = {m["source_file"] for m in methods}

    critical_files = [
        "derek_beach.py",
        "aggregation.py",
        "executors.py",
        "executors_contract.py",
    ]
```

```python

    for critical_file in critical_files:
        found = any(critical_file in sf for sf in source_files)
        assert found, f"Critical file not found in inventory: {critical_file}"


def test_critical_method_patterns(inventory):
    """Verify critical method patterns are present"""
    methods = inventory["methods"]
    canonical_ids = {m["canonical_identifier"] for m in methods}

    patterns_to_check = [
        "derek_beach",
        "aggregation",
        "executors",
    ]

    for pattern in patterns_to_check:
        found = any(pattern in cid.lower() for cid in canonical_ids)
        assert found, f"No methods found matching pattern: {pattern}"


def test_all_roles_present(inventory):
    """Verify all expected roles are present"""
    stats = inventory["statistics"]["by_role"]

    expected_roles = [
        "ingest",
        "processor",
        "analyzer",
        "extractor",
        "score",
        "utility",
        "orchestrator",
        "core",
        "executor",
    ]

    for role in expected_roles:
        assert role in stats, f"Role not found in inventory: {role}"


def test_calibration_flags_set(inventory):
    """Verify calibration flags are properly set"""
    methods = inventory["methods"]

    calibration_count = sum(1 for m in methods if m["requiere_calibracion"])
    parametrization_count = sum(1 for m in methods if m["requiere_parametrizacion"])

    assert calibration_count > 0, "No methods flagged for calibration"
    assert parametrization_count > 0, "No methods flagged for parametrization"


def test_canonical_identifier_format(inventory):
    """Verify canonical identifiers follow module.Class.method format"""
    methods = inventory["methods"]

    for method in methods:
        cid = method["canonical_identifier"]
        parts = cid.split(".")

        assert (
            len(parts) >= MIN_CANONICAL_ID_PARTS
        ), f"Invalid canonical ID format: {cid}"

        if method["class_name"]:
            assert (
                len(parts) >= MIN_CLASS_METHOD_PARTS
            ), f"Class method must have at least {MIN_CLASS_METHOD_PARTS} parts: {cid}"


def test_epistemology_tags_present(inventory):
    """Verify epistemology tags are assigned"""
    methods = inventory["methods"]

    tagged_count = sum(1 for m in methods if m["epistemology_tags"])
    total = len(methods)

    assert tagged_count > 0, "No methods have epistemology tags"

    tag_ratio = tagged_count / total
    assert tag_ratio > MIN_TAG_RATIO, f"Too few methods tagged: {tag_ratio:.2%}"


def test_derek_beach_methods_complete(inventory):
    """Verify derek_beach.py methods are complete"""
    methods = inventory["methods"]
    derek_methods = [m for m in methods if "derek_beach" in m["source_file"]]

    assert (
        len(derek_methods) > MIN_DEREK_BEACH_METHODS
    ), f"Too few derek_beach methods: {len(derek_methods)}"

    required_patterns = ["_format_message", "to_dict", "_load_config", "classify_test"]

    for pattern in required_patterns:
        found = any(pattern in m["method_name"] for m in derek_methods)
        assert found, f"Required derek_beach method not found: {pattern}"


def test_aggregation_classes_present(inventory):
    """Verify aggregation classes are present"""
    methods = inventory["methods"]
    aggregation_methods = [
        m for m in methods if "aggregation" in m["source_file"].lower()
    ]

    assert len(aggregation_methods) > 0, "No aggregation methods found"

    required_classes = [
        "AreaPolicyAggregator",
        "ClusterAggregator",
        "DimensionAggregator",
    ]

    found_classes = {m["class_name"] for m in aggregation_methods if m["class_name"]}

    for req_class in required_classes:
        assert (
            req_class in found_classes
        ), f"Required aggregation class not found: {req_class}"


def test_executor_methods_present(inventory):
    """Verify executor methods are present"""
    methods = inventory["methods"]
    executor_methods = [m for m in methods if "executor" in m["source_file"].lower()]

    assert (
        len(executor_methods) > MIN_EXECUTOR_METHODS
    ), f"Too few executor methods: {len(executor_methods)}"


def test_no_duplicate_canonical_ids(inventory):
    """Verify no duplicate canonical identifiers"""
    methods = inventory["methods"]
    canonical_ids = [m["canonical_identifier"] for m in methods]

    duplicates = [cid for cid in canonical_ids if canonical_ids.count(cid) > 1]

    assert len(duplicates) == 0, f"Duplicate canonical IDs found: {set(duplicates)}"


def test_layer_requirements_complete(inventory):
    """Verify LAYER_REQUIREMENTS table is complete"""
    layer_requirements = inventory["layer_requirements"]

    expected_layers = [
        "ingest",
        "processor",
        "analyzer",
        "extractor",
        "score",
        "utility",
        "orchestrator",
        "core",
        "executor",
    ]

    for layer in expected_layers:
        assert layer in layer_requirements, f"Layer missing from requirements: {layer}"
        assert "description" in layer_requirements[layer]
        assert "typical_patterns" in layer_requirements[layer]


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

```


================================================================================
FILE: tests/test_layer_assignment.py
================================================================================

"""
Test suite for layer assignment system.

VERIFICATION CONDITIONS:
1. All methods must have layer mappings
2. All executors must have exactly 8 layers
3. JSON contains ONLY metadata (no numeric scores outside weights)
4. Sum of weights per executor must equal 1.0

FAILURE CONDITIONS:
- If <30 executors identified: ABORT with 'layer assignment corrupted'
- If any score appears in JSON (outside weights): ABORT with 'layer assignment corrupted'

DEPRECATED: Test assumes outdated canonic_inventorry_methods_layers.json structure.
See tests/DEPRECATED_TESTS.md for details.
"""

import json
from pathlib import Path

import pytest

pytestmark = pytest.mark.obsolete

REPO_ROOT = Path(__file__).parent.parent
CONFIG_FILE = REPO_ROOT / "config" / "canonic_inventorry_methods_layers.json"
EXECUTORS_FILE = (
    REPO_ROOT / "src" / "farfan_pipeline" / "core" / "orchestrator" / "executors.py"
)


class TestLayerAssignment:

    @pytest.fixture
    def inventory_data(self):
        """Load the canonical inventory JSON."""
        if not CONFIG_FILE.exists():
            pytest.fail(f"Config file not found: {CONFIG_FILE}")

        with open(CONFIG_FILE) as f:
            return json.load(f)

    def test_minimum_executor_count(self, inventory_data):
        """Verify at least 30 executors are identified."""
        method_count = len(inventory_data["methods"])
        assert (
            method_count >= 30
        ), f"layer assignment corrupted: Found {method_count} executors, expected 30"

    def test_all_methods_have_layer_mappings(self, inventory_data):
        """Verify every method has layer mappings."""
        for method_id, method_data in inventory_data["methods"].items():
            assert (
                "layers" in method_data
            ), f"layer assignment corrupted: Method {method_id} missing 'layers'"
            assert isinstance(
                method_data["layers"], list
            ), f"layer assignment corrupted: Method {method_id} 'layers' is not a list"
            assert (
                len(method_data["layers"]) > 0
            ), f"layer assignment corrupted: Method {method_id} has empty 'layers'"

    def test_all_executors_have_8_layers(self, inventory_data):
        """Verify all executors have exactly 8 layers."""
        for method_id, method_data in inventory_data["methods"].items():
            if method_data.get("role") == "executor":
                layer_count = len(method_data["layers"])
                assert layer_count == 8, (
                    f"layer assignment corrupted: Executor {method_id} has "
                    f"{layer_count} layers, expected 8"
                )

    def test_no_numeric_scores_in_json(self, inventory_data):
        """Verify JSON contains only metadata, not numeric scores."""

        def check_for_scores(obj, path=""):
            if isinstance(obj, dict):
                for key, value in obj.items():
                    current_path = f"{path}.{key}" if path else key

                    if key in ["weights", "interaction_weights"]:
                        continue

                    if key in ["version", "total_executors"]:
                        continue

                    if isinstance(value, int | float) and key not in [
                        "version",
                        "total_executors",
                    ]:
                        if current_path.endswith(("weights", "interaction_weights")):
                            continue
                        pytest.fail(
                            f"layer assignment corrupted: Found numeric score in JSON "
                            f"at {current_path}={value}"
                        )

                    check_for_scores(value, current_path)

            elif isinstance(obj, list):
                for i, item in enumerate(obj):
                    check_for_scores(item, f"{path}[{i}]")

        check_for_scores(inventory_data)

    def test_weights_sum_to_one(self, inventory_data):
        """Verify sum of weights per executor equals 1.0."""
        for method_id, method_data in inventory_data["methods"].items():
            weights = method_data.get("weights", {})
            interaction_weights = method_data.get("interaction_weights", {})

            total = sum(weights.values()) + sum(interaction_weights.values())

            assert abs(total - 1.0) < 0.01, (
                f"layer assignment corrupted: Weights for {method_id} "
                f"sum to {total:.4f}, expected 1.0"
            )

    def test_required_metadata_fields(self, inventory_data):
        """Verify all methods have required metadata fields."""
        required_fields = ["method_id", "role", "layers", "weights", "aggregator_type"]

        for method_id, method_data in inventory_data["methods"].items():
            for field in required_fields:
                assert field in method_data, (
                    f"layer assignment corrupted: Method {method_id} missing "
                    f"required field '{field}'"
                )

    def test_layer_names_are_valid(self, inventory_data):
        """Verify all layer names match the canonical set."""
        valid_layers = {"@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"}

        for method_id, method_data in inventory_data["methods"].items():
            layers = set(method_data["layers"])
            invalid = layers - valid_layers

            assert not invalid, (
                f"layer assignment corrupted: Method {method_id} has invalid "
                f"layers: {invalid}"
            )

    def test_weights_match_layers(self, inventory_data):
        """Verify weights dict keys match assigned layers."""
        for method_id, method_data in inventory_data["methods"].items():
            layers = set(method_data["layers"])
            weight_keys = set(method_data["weights"].keys())

            assert weight_keys == layers, (
                f"layer assignment corrupted: Method {method_id} weight keys "
                f"{weight_keys} don't match layers {layers}"
            )

    def test_executor_pattern_matching(self):
        """Verify executor pattern D[1-6]Q[1-5] is correctly identified."""
        import re

        with open(EXECUTORS_FILE) as f:
            content = f.read()

        pattern = re.compile(r"class (D([1-6])_Q([1-5])_\w+)\(")
        matches = pattern.findall(content)

        assert len(matches) >= 30, (
            f"layer assignment corrupted: Found {len(matches)} executor classes, "
            f"expected 30"
        )

        for _class_name, dim, question in matches:
            assert dim in "123456", f"Invalid dimension: {dim}"
            assert question in "12345", f"Invalid question: {question}"

    def test_aggregator_type_is_choquet(self, inventory_data):
        """Verify all executors use Choquet aggregator."""
        for method_id, method_data in inventory_data["methods"].items():
            if method_data.get("role") == "executor":
                assert method_data["aggregator_type"] == "choquet", (
                    f"layer assignment corrupted: Executor {method_id} has "
                    f"aggregator_type '{method_data['aggregator_type']}', expected 'choquet'"
                )

    def test_interaction_weights_are_valid(self, inventory_data):
        """Verify interaction weights reference valid layer pairs."""
        for method_id, method_data in inventory_data["methods"].items():
            layers = set(method_data["layers"])

            for pair_key in method_data.get("interaction_weights", {}):
                if "," not in pair_key:
                    pytest.fail(
                        f"layer assignment corrupted: Invalid interaction weight key "
                        f"'{pair_key}' in {method_id}"
                    )

                l1, l2 = pair_key.split(",")
                assert l1 in layers, (
                    f"layer assignment corrupted: Interaction weight {pair_key} in "
                    f"{method_id} references layer {l1} not in assigned layers"
                )
                assert l2 in layers, (
                    f"layer assignment corrupted: Interaction weight {pair_key} in "
                    f"{method_id} references layer {l2} not in assigned layers"
                )

    def test_dimension_and_question_metadata(self, inventory_data):
        """Verify executors have dimension and question metadata."""
        for method_id, method_data in inventory_data["methods"].items():
            if method_data.get("role") == "executor":
                assert (
                    "dimension" in method_data
                ), f"layer assignment corrupted: Executor {method_id} missing 'dimension'"
                assert (
                    "question" in method_data
                ), f"layer assignment corrupted: Executor {method_id} missing 'question'"

                dim = method_data["dimension"]
                question = method_data["question"]

                assert (
                    dim.startswith("D") and dim[1:].isdigit()
                ), f"layer assignment corrupted: Invalid dimension '{dim}' in {method_id}"
                assert (
                    question.startswith("Q") and question[1:].isdigit()
                ), f"layer assignment corrupted: Invalid question '{question}' in {method_id}"


class TestLayerRequirements:
    """Test the LAYER_REQUIREMENTS table definition."""

    def test_layer_requirements_completeness(self):
        """Verify LAYER_REQUIREMENTS covers all specified roles."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected_roles = [
            "ingest",
            "processor",
            "analyzer",
            "score",
            "executor",
            "utility",
            "orchestrator",
            "core",
            "extractor",
        ]

        for role in expected_roles:
            assert (
                role in LAYER_REQUIREMENTS
            ), f"layer assignment corrupted: Role '{role}' missing from LAYER_REQUIREMENTS"

    def test_ingest_layers(self):
        """Verify ingest role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@u", "@m"]
        assert LAYER_REQUIREMENTS["ingest"] == expected

    def test_processor_layers(self):
        """Verify processor role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@u", "@m"]
        assert LAYER_REQUIREMENTS["processor"] == expected

    def test_analyzer_layers(self):
        """Verify analyzer role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
        assert LAYER_REQUIREMENTS["analyzer"] == expected

    def test_score_layers(self):
        """Verify score role has all 8 layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
        assert LAYER_REQUIREMENTS["score"] == expected

    def test_executor_layers(self):
        """Verify executor role has all 8 layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
        assert LAYER_REQUIREMENTS["executor"] == expected

    def test_utility_layers(self):
        """Verify utility role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@m"]
        assert LAYER_REQUIREMENTS["utility"] == expected

    def test_orchestrator_layers(self):
        """Verify orchestrator role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@m"]
        assert LAYER_REQUIREMENTS["orchestrator"] == expected

    def test_core_layers(self):
        """Verify core role has all 8 layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
        assert LAYER_REQUIREMENTS["core"] == expected

    def test_extractor_layers(self):
        """Verify extractor role has correct layers."""
        from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS

        expected = ["@b", "@chain", "@u", "@m"]
        assert LAYER_REQUIREMENTS["extractor"] == expected


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



================================================================================
FILE: tests/test_memory_safety.py
================================================================================

"""
Tests for memory safety guards in executor system.
"""

import pytest
from farfan_pipeline.core.orchestrator.memory_safety import (
    MemorySafetyGuard,
    MemorySafetyConfig,
    ExecutorType,
    ObjectSizeEstimator,
    FallbackStrategy,
    MemoryPressureDetector,
)


class TestObjectSizeEstimator:
    def test_estimate_object_size_simple(self):
        assert ObjectSizeEstimator.estimate_object_size(42) > 0
        assert ObjectSizeEstimator.estimate_object_size("hello") > 0
        assert ObjectSizeEstimator.estimate_object_size([1, 2, 3]) > 0

    def test_estimate_json_size(self):
        assert ObjectSizeEstimator.estimate_json_size(None) == 4
        assert ObjectSizeEstimator.estimate_json_size(True) == 5
        assert ObjectSizeEstimator.estimate_json_size(42) > 0
        assert ObjectSizeEstimator.estimate_json_size("test") > 0
        assert ObjectSizeEstimator.estimate_json_size({"key": "value"}) > 0
        assert ObjectSizeEstimator.estimate_json_size([1, 2, 3]) > 0

    def test_estimate_large_list(self):
        large_list = [{"id": i, "data": "x" * 100} for i in range(1000)]
        size = ObjectSizeEstimator.estimate_json_size(large_list)
        assert size > 100_000


class TestFallbackStrategy:
    def test_sample_list(self):
        items = list(range(1000))
        sampled = FallbackStrategy.sample_list(items, 100)
        assert len(sampled) == 100
        assert all(item in items for item in sampled)

    def test_sample_list_preserve_order(self):
        items = list(range(100))
        sampled = FallbackStrategy.sample_list(items, 10, preserve_order=True)
        assert len(sampled) == 10
        assert sampled == sorted(sampled)

    def test_truncate_string(self):
        long_str = "x" * 1000
        truncated = FallbackStrategy.truncate_string(long_str, 100)
        assert len(truncated) == 100
        assert truncated.endswith("...")

    def test_truncate_dict(self):
        large_dict = {f"key_{i}": i for i in range(1000)}
        truncated = FallbackStrategy.truncate_dict(large_dict, 10)
        assert len(truncated) <= 10

    def test_apply_recursive_truncation(self):
        config = MemorySafetyConfig(
            max_list_elements=10,
            max_string_length=50,
            max_dict_keys=5
        )

        obj = {
            "list": list(range(100)),
            "string": "x" * 1000,
            "dict": {f"key_{i}": i for i in range(100)},
            "nested": {
                "data": list(range(50))
            }
        }

        result, modified = FallbackStrategy.apply_recursive_truncation(obj, config)
        assert modified
        assert len(result["list"]) <= 10
        assert len(result["string"]) <= 50
        assert len(result["dict"]) <= 5
        assert len(result["nested"]["data"]) <= 10


class TestMemorySafetyGuard:
    def test_check_small_object(self):
        guard = MemorySafetyGuard()
        obj = {"id": 1, "name": "test"}
        result, metrics = guard.check_and_process(obj, ExecutorType.GENERIC, "test_obj")

        assert result == obj
        assert metrics.object_size_bytes > 0
        assert metrics.json_size_bytes > 0
        assert not metrics.was_truncated
        assert not metrics.was_sampled

    def test_check_large_object_entity_type(self):
        guard = MemorySafetyGuard(MemorySafetyConfig(
            entity_limit_mb=0.001,
            enable_auto_truncation=True
        ))

        large_obj = {"entities": [{"id": i, "data": "x" * 1000} for i in range(100)]}
        result, metrics = guard.check_and_process(large_obj, ExecutorType.ENTITY, "entities")

        assert metrics.object_size_bytes > guard.config.get_limit_bytes(ExecutorType.ENTITY) or \
               metrics.json_size_bytes > guard.config.get_limit_bytes(ExecutorType.ENTITY) or \
               metrics.was_truncated

    def test_check_large_list(self):
        guard = MemorySafetyGuard(MemorySafetyConfig(
            generic_limit_mb=0.001,
            max_list_elements=10,
            enable_auto_truncation=True
        ))

        large_list = list(range(1000))
        result, metrics = guard.check_and_process(large_list, ExecutorType.GENERIC, "large_list")

        if metrics.was_truncated:
            assert len(result) <= 10

    def test_get_metrics_summary(self):
        guard = MemorySafetyGuard()

        for i in range(5):
            obj = {"id": i, "data": "test" * 100}
            guard.check_and_process(obj, ExecutorType.GENERIC, f"obj_{i}")

        summary = guard.get_metrics_summary()
        assert summary["total_operations"] == 5
        assert summary["avg_object_size_mb"] >= 0
        assert summary["max_object_size_mb"] >= summary["avg_object_size_mb"]


class TestMemoryPressureDetector:
    def test_get_memory_pressure_pct(self):
        pressure = MemoryPressureDetector.get_memory_pressure_pct()
        if pressure is not None:
            assert 0 <= pressure <= 100

    def test_is_under_pressure(self):
        result = MemoryPressureDetector.is_under_pressure(threshold_pct=99.0)
        assert isinstance(result, bool)


class TestMemorySafetyConfig:
    def test_get_limit_bytes(self):
        config = MemorySafetyConfig()

        entity_limit = config.get_limit_bytes(ExecutorType.ENTITY)
        assert entity_limit == 1 * 1024 * 1024

        dag_limit = config.get_limit_bytes(ExecutorType.DAG)
        assert dag_limit == 5 * 1024 * 1024

        causal_limit = config.get_limit_bytes(ExecutorType.CAUSAL_EFFECTS)
        assert causal_limit == 10 * 1024 * 1024

    def test_custom_config(self):
        config = MemorySafetyConfig(
            entity_limit_mb=2.0,
            dag_limit_mb=10.0,
            max_list_elements=500
        )

        assert config.get_limit_bytes(ExecutorType.ENTITY) == 2 * 1024 * 1024
        assert config.get_limit_bytes(ExecutorType.DAG) == 10 * 1024 * 1024
        assert config.max_list_elements == 500



================================================================================
FILE: tests/test_opentelemetry_observability.py
================================================================================

"""Tests for production OpenTelemetry integration.
```

```

Tests cover:
- Configuration and initialization
- Tracing functionality
- Metrics collection
- Executor instrumentation
- Pipeline instrumentation
- Context propagation

DEPRECATED: Test uses outdated farfan_core module path, should use farfan_pipeline.
See tests/DEPRECATED_TESTS.md for details.
"""

import pytest
from unittest.mock import MagicMock, patch

from farfan_core.observability import (
    OpenTelemetryConfig,
    FARFANObservability,
    get_global_observability,
    initialize_observability,
)

pytestmark = pytest.mark.obsolete


class TestOpenTelemetryConfig:
    """Test OpenTelemetry configuration."""

    def test_default_config(self):
        config = OpenTelemetryConfig()
        assert config.service_name == "farfan-pipeline"
        assert config.service_version == "1.0.0"
        assert config.prometheus_port == 9090

    def test_custom_config(self):
        config = OpenTelemetryConfig(
            service_name="test-service",
            service_version="2.0.0",
            prometheus_port=8080,
        )
        assert config.service_name == "test-service"
        assert config.service_version == "2.0.0"
        assert config.prometheus_port == 8080


class TestFARFANObservability:
    """Test main observability class."""

    @patch("farfan_core.observability.opentelemetry_integration.OTEL_AVAILABLE", False)
    def test_initialization_without_otel(self):
        obs = FARFANObservability()
        assert not obs._initialized
        assert obs.get_tracer() is None
        assert obs.get_meter() is None

    def test_start_span_without_otel(self):
        obs = FARFANObservability()
        obs._initialized = False

        with obs.start_span("test_span") as span:
            assert span is None

    def test_trace_executor_without_otel(self):
        obs = FARFANObservability()
        obs._initialized = False

        @obs.trace_executor("D1_Q1_TestExecutor")
        def test_execute(context):
            return {"result": "success"}

        result = test_execute({"test": "data"})
        assert result == {"result": "success"}

    def test_propagate_context_without_otel(self):
        obs = FARFANObservability()
        obs._initialized = False

        carrier = obs.propagate_context()
        assert carrier == {}


class TestGlobalObservability:
    """Test global observability singleton."""

    def test_get_global_observability(self):
        obs1 = get_global_observability()
        obs2 = get_global_observability()
        assert obs1 is obs2

    def test_initialize_observability(self):
        obs = initialize_observability(
            service_name="test-service",
            enable_jaeger=False,
            enable_prometheus=False,
            auto_instrument=False,
        )
        assert isinstance(obs, FARFANObservability)
        assert obs.config.service_name == "test-service"


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



================================================================================
FILE: tests/test_phase6_7_8_integration.py
================================================================================

"""
Integration Tests for Phase 6-7-8 Pipeline
```

```
===========================================

Tests the complete flow:
Phase 6 (Schema Validation) â\206\222 Phase 7 (Task Construction) â\206\222 Phase 8 (Execution Plan)

Verifies:
1. Successful end-to-end integration with valid data
2. Error propagation from Phase 6 â\206\222 Phase 7 â\206\222 Phase 8
3. Field validation and type checking
4. Duplicate detection across phases
5. Non-null constraint enforcement
6. Deterministic identifier generation
7. Integrity hash computation
"""

import pytest
from typing import Any

from src.farfan_pipeline.core.phases.phase6_schema_validation import (
    phase6_schema_validation,
    Phase6SchemaValidationOutput,
    PHASE6_VERSION,
)
from src.farfan_pipeline.core.phases.phase7_task_construction import (
    phase7_task_construction,
    Phase7TaskConstructionOutput,
    PHASE7_VERSION,
    TASK_ID_FORMAT,
)
from src.farfan_pipeline.core.phases.phase8_execution_plan import (
    phase8_execution_plan_assembly,
    Phase8ExecutionPlanOutput,
    PHASE8_VERSION,
)


def create_valid_question(index: int) -> dict[str, Any]:
    """Create a valid question for testing."""
    return {
        "question_id": f"Q{index:03d}",
        "question_global": index,
        "dimension_id": f"D{(index % 6) + 1}",
        "policy_area_id": f"PA{(index % 10) + 1:02d}",
        "base_slot": f"slot_{index}",
        "cluster_id": f"cluster_{index % 4}",
        "patterns": [{"type": "regex", "value": "test.*"}],
        "signals": {"signal_type": "evidence", "strength": 0.8},
        "expected_elements": [{"type": "text", "required": True}],
        "metadata": {"source": "test"},
    }


def create_valid_chunk(index: int) -> dict[str, Any]:
    """Create a valid chunk for testing."""
    return {
        "chunk_id": f"chunk_{index:03d}",
        "policy_area_id": f"PA{(index % 10) + 1:02d}",
        "dimension_id": f"D{(index % 6) + 1}",
        "document_position": index,
        "content": f"Test content for chunk {index}",
        "metadata": {"page": index // 10},
    }


class TestPhase6SchemaValidation:
    """Test Phase 6 schema validation functionality."""

    def test_valid_questions_and_chunks(self):
        """Test successful validation with valid data."""
        questions = [create_valid_question(i) for i in range(10)]
        chunks = [create_valid_chunk(i) for i in range(10)]

        result = phase6_schema_validation(questions, chunks)

        assert result.schema_validation_passed
        assert len(result.validation_errors) == 0
        assert result.question_count == 10
        assert result.chunk_count == 10
        assert len(result.validated_questions) == 10
        assert len(result.validated_chunks) == 10
        assert result.phase6_version == PHASE6_VERSION

    def test_missing_required_field_question(self):
        """Test error handling for missing required field in question."""
        questions = [{"question_id": "Q001"}]
        chunks = []

        result = phase6_schema_validation(questions, chunks)

        assert not result.schema_validation_passed
        assert len(result.validation_errors) > 0
        assert any("question_global" in err for err in result.validation_errors)
        assert any("KeyError" in err for err in result.validation_errors)

    def test_invalid_type_question_global(self):
        """Test error handling for invalid type."""
        question = create_valid_question(0)
        question["question_global"] = "not_an_int"

        result = phase6_schema_validation([question], [])

        assert not result.schema_validation_passed
        assert any("invalid type" in err for err in result.validation_errors)
        assert any("expected int" in err for err in result.validation_errors)

    def test_out_of_range_question_global(self):
        """Test error handling for out-of-range value."""
        question = create_valid_question(0)
        question["question_global"] = 1000

        result = phase6_schema_validation([question], [])

        assert not result.schema_validation_passed
        assert any("not in range" in err for err in result.validation_errors)

    def test_empty_string_field(self):
        """Test error handling for empty string in required field."""
        question = create_valid_question(0)
        question["question_id"] = ""

        result = phase6_schema_validation([question], [])

        assert not result.schema_validation_passed
        assert any("empty string" in err for err in result.validation_errors)

    def test_missing_chunk_content(self):
        """Test error handling for missing chunk content."""
        chunks = [{"chunk_id": "chunk_001"}]

        result = phase6_schema_validation([], chunks)

        assert not result.schema_validation_passed
        assert any("content" in err for err in result.validation_errors)
        assert any("KeyError" in err for err in result.validation_errors)


class TestPhase7TaskConstruction:
    """Test Phase 7 task construction functionality."""

    def test_successful_task_construction(self):
        """Test successful task construction with valid Phase 6 output."""
        questions = [create_valid_question(i) for i in range(5)]
        chunks = [create_valid_chunk(i) for i in range(5)]

        phase6_output = phase6_schema_validation(questions, chunks)
        assert phase6_output.schema_validation_passed

        phase7_output = phase7_task_construction(phase6_output)

        assert phase7_output.construction_passed
        assert len(phase7_output.construction_errors) == 0
        assert phase7_output.task_count == 5
        assert len(phase7_output.tasks) == 5
        assert phase7_output.phase7_version == PHASE7_VERSION

    def test_task_id_format(self):
        """Test task ID generation with zero-padded format."""
        questions = [create_valid_question(i) for i in range(3)]
        chunks = [create_valid_chunk(i) for i in range(3)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)

        assert phase7_output.construction_passed

        for i, task in enumerate(phase7_output.tasks):
            expected_id = TASK_ID_FORMAT.format(
                question_global=i,
                policy_area_id=f"PA{(i % 10) + 1:02d}"
            )
            assert task.task_id == expected_id
            assert len(task.task_id.split("-")[1].split("_")[0]) == 3

    def test_phase6_failure_propagation(self):
        """Test error propagation when Phase 6 fails."""
        phase6_output = Phase6SchemaValidationOutput(
            validated_questions=[],
            validated_chunks=[],
            schema_validation_passed=False,
            validation_errors=["Phase 6 test error"],
            validation_warnings=[],
            question_count=0,
            chunk_count=0,
            validation_timestamp="2025-01-19T00:00:00Z",
        )

        phase7_output = phase7_task_construction(phase6_output)

        assert not phase7_output.construction_passed
        assert len(phase7_output.construction_errors) > 0
        assert any("Phase 6" in err for err in phase7_output.construction_errors)
        assert phase7_output.task_count == 0
        assert len(phase7_output.tasks) == 0

    def test_duplicate_task_id_detection(self):
        """Test duplicate task ID detection."""
        questions = [
            create_valid_question(0),
            create_valid_question(0),
        ]
        chunks = [create_valid_chunk(0)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)

        assert not phase7_output.construction_passed
        assert len(phase7_output.duplicate_task_ids) > 0
        assert any("Duplicate" in err for err in phase7_output.construction_errors)

    def test_missing_field_tracking(self):
        """Test tracking of missing fields by task."""
        from src.farfan_pipeline.core.phases.phase7_task_construction import (
            ValidatedQuestionSchema,
            ValidatedChunkSchema,
            Phase6SchemaValidationOutput,
        )

        question = ValidatedQuestionSchema(
            question_id="",
            question_global=0,
            dimension_id="D1",
            policy_area_id="PA01",
            base_slot="slot_0",
            cluster_id="cluster_0",
            patterns=[],
            signals={},
            expected_elements=[],
            metadata={},
        )

        chunk = ValidatedChunkSchema(
            chunk_id="chunk_000",
            policy_area_id="PA01",
            dimension_id="D1",
            document_position=0,
            content="test",
            metadata={},
        )

        phase6_output = Phase6SchemaValidationOutput(
            validated_questions=[question],
            validated_chunks=[chunk],
            schema_validation_passed=True,
            validation_errors=[],
            validation_warnings=[],
            question_count=1,
            chunk_count=1,
            validation_timestamp="2025-01-19T00:00:00Z",
        )

        phase7_output = phase7_task_construction(phase6_output)

        assert not phase7_output.construction_passed
        assert any("empty string" in err for err in phase7_output.construction_errors)


class TestPhase8ExecutionPlanAssembly:
    """Test Phase 8 execution plan assembly functionality."""

    def test_successful_plan_assembly(self):
        """Test successful execution plan assembly."""
        questions = [create_valid_question(i) for i in range(10)]
        chunks = [create_valid_chunk(i) for i in range(10)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)

        assert phase7_output.construction_passed

        phase8_output = phase8_execution_plan_assembly(phase7_output)

        assert phase8_output.assembly_passed
        assert phase8_output.execution_plan is not None
        assert phase8_output.task_count == 10
        assert len(phase8_output.assembly_errors) == 0
        assert phase8_output.integrity_hash != ""
        assert len(phase8_output.integrity_hash) == 64
        assert phase8_output.phase8_version == PHASE8_VERSION

    def test_phase7_failure_propagation(self):
        """Test error propagation when Phase 7 fails."""
        phase7_output = Phase7TaskConstructionOutput(
            tasks=[],
            task_count=0,
            construction_passed=False,
            construction_errors=["Phase 7 test error"],
            construction_warnings=[],
            duplicate_task_ids=[],
            missing_fields_by_task={},
            construction_timestamp="2025-01-19T00:00:00Z",
            metadata={},
        )

        phase8_output = phase8_execution_plan_assembly(phase7_output)

        assert not phase8_output.assembly_passed
        assert phase8_output.execution_plan is None
        assert len(phase8_output.assembly_errors) > 0
        assert any("Phase 7" in err for err in phase8_output.assembly_errors)

    def test_integrity_hash_determinism(self):
        """Test that integrity hash is deterministic."""
        questions = [create_valid_question(i) for i in range(5)]
        chunks = [create_valid_chunk(i) for i in range(5)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)

        phase8_output1 = phase8_execution_plan_assembly(phase7_output)
        phase8_output2 = phase8_execution_plan_assembly(phase7_output)

        assert phase8_output1.integrity_hash == phase8_output2.integrity_hash

    def test_execution_plan_immutability(self):
        """Test that execution plan tasks are immutable tuple."""
        questions = [create_valid_question(i) for i in range(5)]
        chunks = [create_valid_chunk(i) for i in range(5)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)
        phase8_output = phase8_execution_plan_assembly(phase7_output)

        assert phase8_output.execution_plan is not None
        assert isinstance(phase8_output.execution_plan.tasks, tuple)

        with pytest.raises(AttributeError):
            phase8_output.execution_plan.tasks.append(None)


class TestPhase6_7_8_Integration:
    """Integration tests for the complete Phase 6-7-8 pipeline."""

    def test_end_to_end_success(self):
        """Test successful end-to-end pipeline execution."""
        questions = [create_valid_question(i) for i in range(20)]
        chunks = [create_valid_chunk(i) for i in range(20)]

        phase6_output = phase6_schema_validation(questions, chunks)
        assert phase6_output.schema_validation_passed

        phase7_output = phase7_task_construction(phase6_output)
        assert phase7_output.construction_passed

        phase8_output = phase8_execution_plan_assembly(phase7_output)
        assert phase8_output.assembly_passed
        assert phase8_output.execution_plan is not None

    def test_error_cascade_from_phase6(self):
        """Test that errors cascade correctly from Phase 6 through Phase 8."""
        questions = [{"invalid": "data"}]
        chunks = []

        phase6_output = phase6_schema_validation(questions, chunks)
        assert not phase6_output.schema_validation_passed

        phase7_output = phase7_task_construction(phase6_output)
        assert not phase7_output.construction_passed

        phase8_output = phase8_execution_plan_assembly(phase7_output)
        assert not phase8_output.assembly_passed
        assert phase8_output.execution_plan is None

    def test_deterministic_processing_order(self):
        """Test that processing order is deterministic."""
        questions = [create_valid_question(i) for i in range(10)]
        chunks = [create_valid_chunk(i) for i in range(10)]

        run1_phase6 = phase6_schema_validation(questions, chunks)
        run1_phase7 = phase7_task_construction(run1_phase6)
        run1_phase8 = phase8_execution_plan_assembly(run1_phase7)

        run2_phase6 = phase6_schema_validation(questions, chunks)
        run2_phase7 = phase7_task_construction(run2_phase6)
        run2_phase8 = phase8_execution_plan_assembly(run2_phase7)

        assert len(run1_phase7.tasks) == len(run2_phase7.tasks)
        for task1, task2 in zip(run1_phase7.tasks, run2_phase7.tasks):
            assert task1.task_id == task2.task_id
            assert task1.question_global == task2.question_global

        assert run1_phase8.integrity_hash == run2_phase8.integrity_hash

    def test_metadata_propagation(self):
        """Test that metadata is properly propagated through phases."""
        questions = [create_valid_question(i) for i in range(5)]
        chunks = [create_valid_chunk(i) for i in range(5)]

        phase6_output = phase6_schema_validation(questions, chunks)
        phase7_output = phase7_task_construction(phase6_output)
        phase8_output = phase8_execution_plan_assembly(phase7_output)

        assert phase7_output.metadata["question_count"] == 5
        assert phase7_output.metadata["chunk_count"] == 5

        assert phase8_output.metadata["task_count"] == 5
        assert "integrity_hash_length" in phase8_output.metadata


```