```
tests/test_chain_layer.py

"""
Tests for Chain Layer (@chain) Configuration and Evaluator
"""

import pytest
from orchestration.chain_layer import (
    ChainLayerConfig,
    ChainLayerEvaluator,
    MethodSignature,
    UpstreamOutputs,
    create_default_chain_config
)


class TestChainLayerConfig:
    def test_config_creation_valid(self):
        config = ChainLayerConfig(
            validation_config={
                "strict_mode": False,
                "allow_missing_optional": True,
                "penalize_warnings": True
            },
            score_missing_required=0.0,
            score_missing_critical=0.3,
            score_missing_optional=0.6,
            score_warnings=0.8,
            score_perfect=1.0
        )
        assert config.score_missing_required == 0.0
        assert config.score_missing_critical == 0.3
        assert config.score_perfect == 1.0

    def test_config_scores_range_validation(self):
        with pytest.raises(ValueError, match="must be in range"):
            ChainLayerConfig(
                validation_config={
                    "strict_mode": False,
                    "allow_missing_optional": True,
                    "penalize_warnings": True
                },
                score_missing_required=-0.1,
                score_missing_critical=0.3,
                score_missing_optional=0.6,
                score_warnings=0.8,
                score_perfect=1.0
            )

    def test_config_scores_ordering_validation(self):
        with pytest.raises(ValueError, match="strictly increasing"):
            ChainLayerConfig(
                validation_config={
                    "strict_mode": False,
```

```
                    "allow_missing_optional": True,
                    "penalize_warnings": True
                },
                score_missing_required=0.0,
                score_missing_critical=0.6,
                score_missing_optional=0.3,
                score_warnings=0.8,
                score_perfect=1.0
            )

    def test_default_config(self):
        config = create_default_chain_config()
        assert config.score_missing_required == 0.0
        assert config.score_missing_critical == 0.3
        assert config.score_missing_optional == 0.6
        assert config.score_warnings == 0.8
        assert config.score_perfect == 1.0
        assert config.validation_config["allow_missing_optional"] is True


class TestSignatureValidation:
    def test_all_required_inputs_available(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text", "dimension_id"],
            optional_inputs=["context"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text", "dimension_id", "metadata"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["score"] == 1.0
        assert result["validation_status"] == "perfect"
        assert len(result["missing_required"]) == 0

    def test_missing_required_input(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text", "dimension_id"],
            optional_inputs=["context"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
```

```
            available_outputs={"policy_text"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["score"] == 0.0
        assert result["validation_status"] == "failed_missing_required"
        assert "dimension_id" in result["missing_required"]

    def test_missing_critical_optional(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text"],
            optional_inputs=["context", "metadata"],
            critical_optional=["context"],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text", "metadata"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["score"] == 0.3
        assert result["validation_status"] == "failed_missing_critical"
        assert "context" in result["missing_critical"]

    def test_missing_optional_only(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text"],
            optional_inputs=["context", "metadata"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["score"] == 1.0
        assert result["validation_status"] == "perfect"
        assert len(result["missing_optional"]) == 2

    def test_missing_optional_with_penalty(self):
        config = ChainLayerConfig(
            validation_config={
```

```
                "strict_mode": False,
                "allow_missing_optional": False,
                "penalize_warnings": True
            },
            score_missing_required=0.0,
            score_missing_critical=0.3,
            score_missing_optional=0.6,
            score_warnings=0.8,
            score_perfect=1.0
        )
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text"],
            optional_inputs=["context", "metadata"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["score"] == 0.6
        assert result["validation_status"] == "passed_missing_optional"

    def test_available_ratio_calculation(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text", "dimension_id"],
            optional_inputs=["context", "metadata"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text", "context"},
            output_types={}
        )

        result = evaluator.validate_signature_against_upstream(signature, upstream)
        assert result["available_ratio"] == 0.5


class TestChainLayerEvaluation:
    def test_perfect_chain(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
```

```
            required_inputs=["policy_text"],
            optional_inputs=["metadata"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text", "metadata"},
            output_types={"policy_text": "str", "metadata": "dict"}
        )

        result = evaluator.evaluate(signature, upstream)
        assert result["chain_score"] == 1.0
        assert result["validation_status"] == "perfect"
        assert len(result["missing_required"]) == 0
        assert len(result["warnings"]) == 0

    def test_failed_chain_missing_required(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text", "dimension_id"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"metadata"},
            output_types={}
        )

        result = evaluator.evaluate(signature, upstream)
        assert result["chain_score"] == 0.0
        assert result["validation_status"] == "failed_missing_required"
        assert len(result["missing_required"]) == 2

    def test_chain_with_warnings(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signature = MethodSignature(
            required_inputs=["policy_text"],
            optional_inputs=["context", "metadata"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )
        upstream = UpstreamOutputs(
            available_outputs={"policy_text"},
            output_types={}
        )
```

```
        result = evaluator.evaluate(signature, upstream)
        assert "config" in result
        assert "score_thresholds" in result
        assert result["available_ratio"] < 1.0


class TestChainSequenceEvaluation:
    def test_simple_sequence(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signatures = [
            ("method1", MethodSignature(
                required_inputs=["input_data"],
                optional_inputs=[],
                critical_optional=[],
                output_type="dict",
                output_range=None
            )),
            ("method2", MethodSignature(
                required_inputs=["method1"],
                optional_inputs=[],
                critical_optional=[],
                output_type="dict",
                output_range=None
            )),
            ("method3", MethodSignature(
                required_inputs=["method1", "method2"],
                optional_inputs=[],
                critical_optional=[],
                output_type="dict",
                output_range=None
            ))
        ]
        initial_inputs = {"input_data"}

        result = evaluator.evaluate_chain_sequence(signatures, initial_inputs)
        assert result["sequence_score"] == 1.0
        assert result["failed_methods"] == 0
        assert result["total_methods"] == 3
        assert "method1" in result["final_available_outputs"]
        assert "method2" in result["final_available_outputs"]
        assert "method3" in result["final_available_outputs"]

    def test_sequence_with_failures(self):
        config = create_default_chain_config()
        evaluator = ChainLayerEvaluator(config)

        signatures = [
            ("method1", MethodSignature(
                required_inputs=["missing_input"],
                optional_inputs=[],
                critical_optional=[],
                output_type="dict",
```

```
            output_range=None
        )),
        ("method2", MethodSignature(
            required_inputs=["method1"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        ))
    ]
    initial_inputs = {"input_data"}

    result = evaluator.evaluate_chain_sequence(signatures, initial_inputs)
    assert result["sequence_score"] < 1.0
    assert result["failed_methods"] >= 1

def test_sequence_with_branching(self):
    config = create_default_chain_config()
    evaluator = ChainLayerEvaluator(config)

    signatures = [
        ("method1", MethodSignature(
            required_inputs=["input_data"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )),
        ("method2a", MethodSignature(
            required_inputs=["method1"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )),
        ("method2b", MethodSignature(
            required_inputs=["method1"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )),
        ("method3", MethodSignature(
            required_inputs=["method2a", "method2b"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        ))
    ]
    initial_inputs = {"input_data"}

    result = evaluator.evaluate_chain_sequence(signatures, initial_inputs)
    assert result["sequence_score"] == 1.0
```

```
    assert result["failed_methods"] == 0
    assert "method3" in result["final_available_outputs"]

def test_sequence_with_critical_optional(self):
    config = create_default_chain_config()
    evaluator = ChainLayerEvaluator(config)

    signatures = [
        ("method1", MethodSignature(
            required_inputs=["input_data"],
            optional_inputs=["context"],
            critical_optional=["context"],
            output_type="dict",
            output_range=None
        )),
        ("method2", MethodSignature(
            required_inputs=["method1"],
            optional_inputs=[],
            critical_optional=[],
            output_type="dict",
            output_range=None
        ))
    ]
    initial_inputs = {"input_data"}

    result = evaluator.evaluate_chain_sequence(signatures, initial_inputs)
    assert result["sequence_score"] < 1.0
    assert result["method_results"][0]["score"] == 0.3
    assert result["method_results"][0]["status"] == "failed_missing_critical"

def test_empty_sequence(self):
    config = create_default_chain_config()
    evaluator = ChainLayerEvaluator(config)

    result = evaluator.evaluate_chain_sequence([], {"input_data"})
    assert result["sequence_score"] == 0.0
    assert result["total_methods"] == 0
    assert result["failed_methods"] == 0
```

tests/test_configuration_hash_determinism.py

```
"""
Test Suite for Configuration Hash Determinism and Runtime Mode Behavior

This test suite validates:
1. Hash determinism across dict ordering permutations
2. RuntimeMode behavioral enforcement (PROD/DEV/EXPLORATORY)
3. Configuration validation and fail-fast mechanisms
4. Regression prevention for hash computation

Author: F.A.R.F.A.N Test Suite
Date: 2025-12-17
"""

import hashlib
import json
import os
import sys
from itertools import permutations
from pathlib import Path
from types import MappingProxyType
from typing import Any

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

import pytest

from canonic_phases.Phase_zero.runtime_config import (
    ConfigurationError,
    RuntimeConfig,
    RuntimeMode,
    reset_runtime_config,
)


def compute_hash(data: dict[str, Any]) -> str:
    """Compute SHA256 hash of dict using canonical JSON serialization."""
      json_str = json.dumps(data, sort_keys=True, ensure_ascii=False, separators=(",",
":"))
    return hashlib.sha256(json_str.encode("utf-8")).hexdigest()


class TestHashDeterminism:
    """Test suite for configuration hash determinism."""

    def test_same_content_same_hash(self):
        """Test identical content produces identical hash."""
        data1 = {"a": 1, "b": 2, "c": 3}
        data2 = {"a": 1, "b": 2, "c": 3}

        hash1 = compute_hash(data1)
        hash2 = compute_hash(data2)
```

```
        assert hash1 == hash2
        print(f"  ? Identical content produces same hash: {hash1[:16]}...")

    def test_different_key_order_same_hash(self):
        """Test different key ordering produces same hash due to sort_keys."""
        data1 = {"z": 26, "a": 1, "m": 13}
        data2 = {"a": 1, "m": 13, "z": 26}
        data3 = {"m": 13, "z": 26, "a": 1}

        hash1 = compute_hash(data1)
        hash2 = compute_hash(data2)
        hash3 = compute_hash(data3)

        assert hash1 == hash2 == hash3
        print(f"  ? Different key orders produce same hash: {hash1[:16]}...")

    def test_nested_dict_key_order_same_hash(self):
        """Test nested dicts with different key orders produce same hash."""
        data1 = {
            "outer": {"z": 3, "a": 1, "m": 2},
            "list": [{"b": 2, "a": 1}, {"d": 4, "c": 3}]
        }
        data2 = {
            "outer": {"a": 1, "m": 2, "z": 3},
            "list": [{"a": 1, "b": 2}, {"c": 3, "d": 4}]
        }

        hash1 = compute_hash(data1)
        hash2 = compute_hash(data2)

        assert hash1 == hash2
               print(f"   ? Nested dicts with different orders produce same hash:
{hash1[:16]}...")

    def test_all_permutations_same_hash(self):
        """Test all permutations of keys produce same hash."""
        base_data = {"key1": 10, "key2": 20, "key3": 30, "key4": 40}

        # Create permutations by reconstructing dict with different key orders
        permuted_dicts = [
            {"key2": 20, "key1": 10, "key3": 30, "key4": 40},
            {"key3": 30, "key4": 40, "key1": 10, "key2": 20},
            {"key4": 40, "key3": 30, "key2": 20, "key1": 10},
            {"key1": 10, "key3": 30, "key2": 20, "key4": 40},
        ]

        hashes = {compute_hash(d) for d in [base_data] + permuted_dicts}

        assert len(hashes) == 1
        print(f"  ? All 5 key orderings produce identical hash")

    def test_monolith_structure_hash_stability(self):
        """Test realistic monolith structure produces stable hash."""
        monolith = {
```

```
        "blocks": {
            "micro_questions": [
                {"id": "Q001", "text": "Question 1"},
                {"id": "Q002", "text": "Question 2"},
            ],
            "meso_questions": [
                {"id": "M001", "text": "Meso 1"},
            ],
            "macro_question": {"id": "MACRO", "text": "Macro question"},
        },
        "metadata": {
            "version": "1.0.0",
            "created": "2025-12-17",
        }
    }

    hash1 = compute_hash(monolith)

    # Reorder top-level keys
    monolith_reordered = {
        "metadata": monolith["metadata"],
        "blocks": monolith["blocks"],
    }
    hash2 = compute_hash(monolith_reordered)

    assert hash1 == hash2
    print(f"  ? Monolith structure produces stable hash: {hash1[:16]}...")

def test_mapping_proxy_normalization(self):
    """Test MappingProxyType is normalized consistently."""
    data_dict = {"a": 1, "b": 2}
    data_proxy = MappingProxyType(data_dict)

    hash_dict = compute_hash(dict(data_dict))
    hash_proxy = compute_hash(dict(data_proxy))

    assert hash_dict == hash_proxy
    print("  ? MappingProxyType normalized to same hash as dict")

def test_unicode_handling_deterministic(self):
    """Test Unicode content produces deterministic hash."""
    data1 = {"text": "Análisis de políticas públicas"}
    data2 = {"text": "Análisis de políticas públicas"}

    hash1 = compute_hash(data1)
    hash2 = compute_hash(data2)

    assert hash1 == hash2
    print(f"  ? Unicode content produces deterministic hash: {hash1[:16]}...")

def test_empty_structures_deterministic(self):
    """Test empty structures produce deterministic hashes."""
    empty_dict = {}
    empty_list = []
```

```
        hash1 = compute_hash(empty_dict)
        hash2 = compute_hash(empty_dict)

        assert hash1 == hash2
        print("  ? Empty structures produce deterministic hash")

    def test_large_monolith_hash_performance(self):
        """Test hash computation completes quickly for large monoliths."""
        import time

        large_monolith = {
            "blocks": {
                "micro_questions": [
                      {"id": f"Q{i:03d}", "text": f"Question {i}", "metadata": {"index":
i}}
                    for i in range(300)
                ],
            }
        }

        start = time.perf_counter()
        hash_val = compute_hash(large_monolith)
        duration = time.perf_counter() - start

        assert duration < 0.1  # Should complete in under 100ms
        assert len(hash_val) == 64  # SHA256 produces 64 hex chars
        print(f"  ? Large monolith (300 questions) hashed in {duration*1000:.2f}ms")


class TestRuntimeModeBehavior:
    """Test suite for runtime mode behavioral enforcement."""

    def setup_method(self):
        """Reset config before each test."""
        reset_runtime_config()
        for key in list(os.environ.keys()):
             if key.startswith(("SAAAAAA_", "ALLOW_", "STRICT_", "PHASE_", "EXPECTED_",
"PREFERRED_")):
                del os.environ[key]

    def teardown_method(self):
        """Cleanup after each test."""
        reset_runtime_config()

    def test_prod_mode_creates_verified_status(self):
        """Test PROD mode sets verification_status='verified'."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        assert config.mode == RuntimeMode.PROD
        assert config.is_strict_mode()
        print("  ? PROD mode enables strict enforcement")
```

```
def test_dev_mode_creates_development_status(self):
    """Test DEV mode sets verification_status='development'."""
    os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
    config = RuntimeConfig.from_env()

    assert config.mode == RuntimeMode.DEV
    assert not config.is_strict_mode()
    print("  ? DEV mode disables strict enforcement")

def test_exploratory_mode_creates_experimental_status(self):
    """Test EXPLORATORY mode sets verification_status='experimental'."""
    os.environ["SAAAAAA_RUNTIME_MODE"] = "exploratory"
    config = RuntimeConfig.from_env()

    assert config.mode == RuntimeMode.EXPLORATORY
    assert not config.is_strict_mode()
    print("  ? EXPLORATORY mode disables strict enforcement")

def test_prod_mode_rejects_partial_results_flags(self):
    """Test PROD mode rejects flags that enable partial results."""
    os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
    os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"

    with pytest.raises(ConfigurationError, match="Illegal configuration"):
        RuntimeConfig.from_env()

    print("  ? PROD mode rejects ALLOW_AGGREGATION_DEFAULTS")

def test_dev_mode_allows_partial_results_flags(self):
    """Test DEV mode allows flags for partial results."""
    os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
    os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
    os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"

    config = RuntimeConfig.from_env()

    assert config.allow_aggregation_defaults
    assert config.allow_dev_ingestion_fallbacks
    print("  ? DEV mode allows partial result flags")

def test_exploratory_mode_allows_all_flags(self):
    """Test EXPLORATORY mode allows all experimental flags."""
    os.environ["SAAAAAA_RUNTIME_MODE"] = "exploratory"
    os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
    os.environ["ALLOW_EXECUTION_ESTIMATES"] = "true"
    os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"

    config = RuntimeConfig.from_env()

    assert config.allow_aggregation_defaults
    assert config.allow_execution_estimates
    assert config.allow_dev_ingestion_fallbacks
    print("  ? EXPLORATORY mode allows all experimental flags")
```

```
    def test_invalid_runtime_mode_fails_fast(self):
        """Test invalid runtime mode value causes immediate failure."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "invalid_mode"

        with pytest.raises(ConfigurationError, match="Invalid SAAAAAA_RUNTIME_MODE"):
            RuntimeConfig.from_env()

        print("  ? Invalid runtime mode fails fast with clear error")

    def test_strict_calibration_property(self):
        """Test strict_calibration property reflects PROD mode correctly."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        assert config.strict_calibration
        print("  ? PROD mode enforces strict calibration")

        os.environ["ALLOW_MISSING_BASE_WEIGHTS"] = "true"
        reset_runtime_config()

        with pytest.raises(ConfigurationError):
            RuntimeConfig.from_env()

        print("  ? PROD + ALLOW_MISSING_BASE_WEIGHTS rejected")

    def test_dev_mode_relaxed_calibration(self):
        """Test DEV mode allows relaxed calibration."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        os.environ["ALLOW_MISSING_BASE_WEIGHTS"] = "true"
        config = RuntimeConfig.from_env()

        assert not config.strict_calibration
        assert config.allow_missing_base_weights
        print("  ? DEV mode allows relaxed calibration")


class TestRegressionPrevention:
    """Test suite for preventing hash and mode regressions."""

    def setup_method(self):
        """Reset config before each test."""
        reset_runtime_config()
        for key in list(os.environ.keys()):
              if key.startswith(("SAAAAAA_", "ALLOW_", "STRICT_", "PHASE_", "EXPECTED_",
"PREFERRED_")):
                  del os.environ[key]

    def teardown_method(self):
        """Cleanup after each test."""
        reset_runtime_config()

    def test_reference_hash_unchanged(self):
        """Test reference monolith produces known hash (regression detection)."""
        reference_monolith = {
```

```
            "blocks": {
                "micro_questions": [
                    {"id": "Q001", "text": "Test question 1"},
                    {"id": "Q002", "text": "Test question 2"},
                ],
                "meso_questions": [],
                "macro_question": {},
            },
            "version": "1.0.0",
        }

        hash_val = compute_hash(reference_monolith)

        # This hash should remain stable across all code changes
        # If this test fails, hash computation logic has changed
        assert len(hash_val) == 64
        assert all(c in "0123456789abcdef" for c in hash_val)
        print(f"  ? Reference hash format valid: {hash_val[:16]}...")

    def test_runtime_config_defaults_stable(self):
        """Test RuntimeConfig default values remain stable."""
        config = RuntimeConfig.from_env()

        assert config.mode == RuntimeMode.PROD
        assert not config.allow_contradiction_fallback
        assert not config.allow_validator_disable
        assert not config.allow_execution_estimates
        assert not config.allow_aggregation_defaults
        assert config.allow_hash_fallback  # Operational flexibility
        print("  ? RuntimeConfig defaults remain stable")

    def test_runtime_mode_enum_values_stable(self):
        """Test RuntimeMode enum values remain unchanged."""
        assert RuntimeMode.PROD.value == "prod"
        assert RuntimeMode.DEV.value == "dev"
        assert RuntimeMode.EXPLORATORY.value == "exploratory"
        print("  ? RuntimeMode enum values stable")


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

```
tests/test_congruence_chain_integration.py

"""
Integration Tests for Congruence (@C) and Chain (@chain) Layer Evaluators

Tests the interaction between congruence and chain layer evaluators
in realistic pipeline scenarios.
"""

import pytest
from orchestration.congruence_layer import (
    CongruenceLayerEvaluator,
    OutputRangeSpec,
    SemanticTagSet,
    FusionRule,
    create_default_congruence_config
)
from orchestration.chain_layer import (
    ChainLayerEvaluator,
    MethodSignature,
    UpstreamOutputs,
    create_default_chain_config
)


class TestCongruenceChainIntegration:
    def test_executor_ensemble_with_chaining(self):
        """Test executor ensemble evaluation with chain validation."""
        congruence_config = create_default_congruence_config()
        chain_config = create_default_chain_config()

        congruence_eval = CongruenceLayerEvaluator(congruence_config)
        chain_eval = ChainLayerEvaluator(chain_config)

        executor_specs = {
            "CausalExtractor": {
                "signature": MethodSignature(
                    required_inputs=["policy_text", "dimension_id"],
                    optional_inputs=["context"],
                    critical_optional=["context"],
                    output_type="dict",
                    output_range=[0.0, 1.0]
                ),
                "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
                "tags": SemanticTagSet(tags={"causal", "temporal"}, description=None)
            },
            "GoalAnalyzer": {
                "signature": MethodSignature(
                    required_inputs=["policy_text", "dimension_id"],
                    optional_inputs=["context"],
                    critical_optional=["context"],
                    output_type="dict",
                    output_range=[0.0, 1.0]
                ),
```

```
                "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
                                "tags":  SemanticTagSet(tags={"causal",  "goal-oriented"},
description=None)
            }
        }

        upstream = UpstreamOutputs(
            available_outputs={"policy_text", "dimension_id", "context"},
                  output_types={"policy_text":  "str",  "dimension_id":  "str",  "context":
"dict"}
        )

        chain_scores = {}
        for name, spec in executor_specs.items():
            result = chain_eval.evaluate(spec["signature"], upstream)
            chain_scores[name] = result["chain_score"]

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
            description="Executor ensemble fusion"
        )

        congruence_result = congruence_eval.evaluate(
            executor_specs["CausalExtractor"]["range"],
            executor_specs["GoalAnalyzer"]["range"],
            executor_specs["CausalExtractor"]["tags"],
            executor_specs["GoalAnalyzer"]["tags"],
            fusion
        )

        assert all(score == 1.0 for score in chain_scores.values())
        assert congruence_result["C_play"] > 0.0
        assert congruence_result["c_scale"] == 1.0

    def test_pipeline_validation_sequence(self):
        """Test full pipeline validation with both evaluators."""
        chain_config = create_default_chain_config()
        chain_eval = ChainLayerEvaluator(chain_config)

        congruence_config = create_default_congruence_config()
        congruence_eval = CongruenceLayerEvaluator(congruence_config)

        method_signatures = [
            ("PolicyIngestor", MethodSignature(
                required_inputs=["raw_document"],
                optional_inputs=[],
                critical_optional=[],
                output_type="str",
                output_range=None
            )),
            ("PolicyProcessor", MethodSignature(
                required_inputs=["PolicyIngestor"],
```

```
            optional_inputs=["config"],
            critical_optional=[],
            output_type="dict",
            output_range=None
        )),
        ("CausalExtractor", MethodSignature(
            required_inputs=["PolicyProcessor"],
            optional_inputs=["semantic_model"],
            critical_optional=["semantic_model"],
            output_type="dict",
            output_range=[0.0, 1.0]
        ))
    ]

    chain_result = chain_eval.evaluate_chain_sequence(
        method_signatures,
        {"raw_document"}
    )

    assert chain_result["total_methods"] == 3
    assert chain_result["failed_methods"] >= 1

    processor_range = OutputRangeSpec(min=0.0, max=1.0, output_type="dict")
    extractor_range = OutputRangeSpec(min=0.0, max=1.0, output_type="dict")
    processor_tags = SemanticTagSet(tags={"text", "processing"}, description=None)
    extractor_tags = SemanticTagSet(tags={"causal", "temporal"}, description=None)

    fusion = FusionRule(
        rule_type="transformation",
        operator="normalize",
        is_valid=True,
        description=None
    )

    congruence_result = congruence_eval.evaluate(
        extractor_range, processor_range,
        extractor_tags, processor_tags,
        fusion
    )

    assert congruence_result["c_scale"] == 1.0
    assert congruence_result["c_sem"] == 0.0

def test_ensemble_congruence_with_chain_failures(self):
    """Test congruence evaluation when chain validation fails."""
    chain_config = create_default_chain_config()
    chain_eval = ChainLayerEvaluator(chain_config)

    congruence_config = create_default_congruence_config()
    congruence_eval = CongruenceLayerEvaluator(congruence_config)

    signature = MethodSignature(
        required_inputs=["missing_input"],
        optional_inputs=[],
```

```
        critical_optional=[],
        output_type="dict",
        output_range=[0.0, 1.0]
    )

    upstream = UpstreamOutputs(
        available_outputs={"available_input"},
        output_types={}
    )

    chain_result = chain_eval.evaluate(signature, upstream)
    assert chain_result["chain_score"] == 0.0

    range1 = OutputRangeSpec(min=0.0, max=1.0, output_type="dict")
    range2 = OutputRangeSpec(min=0.0, max=1.0, output_type="dict")
    tags1 = SemanticTagSet(tags={"test"}, description=None)
    tags2 = SemanticTagSet(tags={"test"}, description=None)
    fusion = FusionRule(
        rule_type="aggregation",
        operator="sum",
        is_valid=True,
        description=None
    )

    congruence_result = congruence_eval.evaluate(
        range1, range2, tags1, tags2, fusion
    )

    assert congruence_result["C_play"] == 1.0

def test_mixed_scores_aggregation(self):
    """Test aggregation of mixed chain and congruence scores."""
    chain_config = create_default_chain_config()
    chain_eval = ChainLayerEvaluator(chain_config)

    congruence_config = create_default_congruence_config()
    congruence_eval = CongruenceLayerEvaluator(congruence_config)

    methods = [
        {
            "signature": MethodSignature(
                required_inputs=["input"],
                optional_inputs=["context"],
                critical_optional=["context"],
                output_type="dict",
                output_range=[0.0, 1.0]
            ),
            "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
            "tags": SemanticTagSet(tags={"causal", "temporal"}, description=None)
        },
        {
            "signature": MethodSignature(
                required_inputs=["input"],
                optional_inputs=["metadata"],
```

```
                critical_optional=[],
                output_type="dict",
                output_range=[0.0, 1.0]
            ),
            "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
            "tags": SemanticTagSet(tags={"causal", "numeric"}, description=None)
        }
    ]

    upstream = UpstreamOutputs(
        available_outputs={"input"},
        output_types={"input": "str"}
    )

    chain_scores = []
    for method in methods:
        result = chain_eval.evaluate(method["signature"], upstream)
        chain_scores.append(result["chain_score"])

    fusion = FusionRule(
        rule_type="aggregation",
        operator="weighted_avg",
        is_valid=True,
        description=None
    )

    congruence_result = congruence_eval.evaluate(
        methods[0]["range"], methods[1]["range"],
        methods[0]["tags"], methods[1]["tags"],
        fusion
    )

    assert chain_scores[0] == 0.3
    assert chain_scores[1] == 1.0

    assert 0.0 < congruence_result["C_play"] < 1.0

    w_chain = 0.13
    w_congruence = 0.08

    combined_score = (
        (chain_scores[0] * w_chain) +
        (chain_scores[1] * w_chain) +
        (congruence_result["C_play"] * w_congruence)
    ) / (2 * w_chain + w_congruence)

    assert 0.0 <= combined_score <= 1.0

def test_full_layer_evaluation_ensemble(self):
    """Test complete layer evaluation for an executor ensemble."""
    chain_config = create_default_chain_config()
    chain_eval = ChainLayerEvaluator(chain_config)

    congruence_config = create_default_congruence_config()
```

```
congruence_eval = CongruenceLayerEvaluator(congruence_config)

executors = [
    {
        "name": "D1_Q1_Executor",
        "signature": MethodSignature(
            required_inputs=["policy_text", "dimension_id", "question_id"],
            optional_inputs=["context", "metadata"],
            critical_optional=["context"],
            output_type="dict",
            output_range=[0.0, 1.0]
        ),
        "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
        "tags": SemanticTagSet(
            tags={"causal", "temporal", "dimension_1"},
            description="D1-Q1 executor"
        )
    },
    {
        "name": "D1_Q2_Executor",
        "signature": MethodSignature(
            required_inputs=["policy_text", "dimension_id", "question_id"],
            optional_inputs=["context", "metadata"],
            critical_optional=["context"],
            output_type="dict",
            output_range=[0.0, 1.0]
        ),
        "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
        "tags": SemanticTagSet(
            tags={"causal", "numeric", "dimension_1"},
            description="D1-Q2 executor"
        )
    },
    {
        "name": "D2_Q1_Executor",
        "signature": MethodSignature(
            required_inputs=["policy_text", "dimension_id", "question_id"],
            optional_inputs=["context", "metadata"],
            critical_optional=["context"],
            output_type="dict",
            output_range=[0.0, 1.0]
        ),
        "range": OutputRangeSpec(min=0.0, max=1.0, output_type="dict"),
        "tags": SemanticTagSet(
            tags={"goal-oriented", "strategic", "dimension_2"},
            description="D2-Q1 executor"
        )
    }
]

upstream = UpstreamOutputs(
    available_outputs={"policy_text", "dimension_id", "question_id", "context"},
    output_types={
        "policy_text": "str",
```

```
        "dimension_id": "str",
        "question_id": "str",
        "context": "dict"
    }
)

chain_results = {}
for executor in executors:
    result = chain_eval.evaluate(executor["signature"], upstream)
    chain_results[executor["name"]] = {
        "chain_score": result["chain_score"],
        "status": result["validation_status"]
    }

fusion = FusionRule(
    rule_type="aggregation",
    operator="weighted_avg",
    is_valid=True,
    description="Dimension-level fusion"
)

congruence_scores = {}
for i in range(len(executors)):
    for j in range(i + 1, len(executors)):
        pair_key = f"{executors[i]['name']} <-> {executors[j]['name']}"
        result = congruence_eval.evaluate(
            executors[i]["range"], executors[j]["range"],
            executors[i]["tags"], executors[j]["tags"],
            fusion
        )
        congruence_scores[pair_key] = result["C_play"]

assert all(r["chain_score"] == 1.0 for r in chain_results.values())
assert all(r["status"] == "perfect" for r in chain_results.values())

d1_congruence = congruence_scores["D1_Q1_Executor <-> D1_Q2_Executor"]
cross_dim_congruence = congruence_scores["D1_Q1_Executor <-> D2_Q1_Executor"]

assert d1_congruence > cross_dim_congruence
```

tests/test_congruence_layer.py

```
"""
Tests for Congruence Layer (@C) Configuration and Evaluator
"""

import pytest
from orchestration.congruence_layer import (
    CongruenceLayerConfig,
    CongruenceLayerEvaluator,
    OutputRangeSpec,
    SemanticTagSet,
    FusionRule,
    create_default_congruence_config
)


class TestCongruenceLayerConfig:
    def test_config_creation_valid(self):
        config = CongruenceLayerConfig(
            w_scale=0.4,
            w_semantic=0.35,
            w_fusion=0.25,
            requirements={
                "require_output_range_compatibility": True,
                "require_semantic_alignment": True,
                "require_fusion_validity": True
            },
            thresholds={
                "min_jaccard_similarity": 0.3,
                "max_range_mismatch_ratio": 0.5,
                "min_fusion_validity_score": 0.6
            }
        )
        assert config.w_scale == 0.4
        assert config.w_semantic == 0.35
        assert config.w_fusion == 0.25

    def test_config_weights_sum_validation(self):
        with pytest.raises(ValueError, match="must sum to 1.0"):
            CongruenceLayerConfig(
                w_scale=0.5,
                w_semantic=0.3,
                w_fusion=0.1,
                requirements={
                    "require_output_range_compatibility": True,
                    "require_semantic_alignment": True,
                    "require_fusion_validity": True
                },
                thresholds={
                    "min_jaccard_similarity": 0.3,
                    "max_range_mismatch_ratio": 0.5,
                    "min_fusion_validity_score": 0.6
                }
```

```
            )

    def test_config_negative_weights_validation(self):
        with pytest.raises(ValueError, match="must be non-negative"):
            CongruenceLayerConfig(
                w_scale=0.6,
                w_semantic=0.5,
                w_fusion=-0.1,
                requirements={
                    "require_output_range_compatibility": True,
                    "require_semantic_alignment": True,
                    "require_fusion_validity": True
                },
                thresholds={
                    "min_jaccard_similarity": 0.3,
                    "max_range_mismatch_ratio": 0.5,
                    "min_fusion_validity_score": 0.6
                }
            )

    def test_default_config(self):
        config = create_default_congruence_config()
        assert config.w_scale == 0.4
        assert config.w_semantic == 0.35
        assert config.w_fusion == 0.25
        assert config.requirements["require_output_range_compatibility"] is True
        assert config.thresholds["min_jaccard_similarity"] == 0.3


class TestOutputScaleCompatibility:
    def test_perfect_overlap(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        upstream = OutputRangeSpec(min=0.0, max=1.0, output_type="float")

        score = evaluator.evaluate_output_scale_compatibility(current, upstream)
        assert score == 1.0

    def test_partial_overlap(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        upstream = OutputRangeSpec(min=0.5, max=1.5, output_type="float")

        score = evaluator.evaluate_output_scale_compatibility(current, upstream)
        assert 0.0 < score < 1.0

    def test_no_overlap(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)
```

```
        current = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        upstream = OutputRangeSpec(min=2.0, max=3.0, output_type="float")

        score = evaluator.evaluate_output_scale_compatibility(current, upstream)
        assert score == 0.0

    def test_type_mismatch(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        upstream = OutputRangeSpec(min=0.0, max=1.0, output_type="int")

        score = evaluator.evaluate_output_scale_compatibility(current, upstream)
        assert score == 0.0

    def test_zero_span(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = OutputRangeSpec(min=0.5, max=0.5, output_type="float")
        upstream = OutputRangeSpec(min=0.0, max=1.0, output_type="float")

        score = evaluator.evaluate_output_scale_compatibility(current, upstream)
        assert score == 0.0


class TestSemanticAlignment:
    def test_perfect_match(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

                    current   =   SemanticTagSet(tags={"causal",   "temporal",   "numeric"},
description=None)
                   upstream   =   SemanticTagSet(tags={"causal",   "temporal",   "numeric"},
description=None)

        score = evaluator.evaluate_semantic_alignment(current, upstream)
        assert score == 1.0

    def test_partial_overlap(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = SemanticTagSet(tags={"causal", "temporal"}, description=None)
        upstream = SemanticTagSet(tags={"causal", "numeric"}, description=None)

        score = evaluator.evaluate_semantic_alignment(current, upstream)
        assert 0.0 < score < 1.0
        assert abs(score - (1.0 / 3.0)) < 0.01

    def test_no_overlap(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)
```

```
        current = SemanticTagSet(tags={"causal"}, description=None)
        upstream = SemanticTagSet(tags={"numeric"}, description=None)

        score = evaluator.evaluate_semantic_alignment(current, upstream)
        assert score == 0.0

    def test_empty_tags(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current = SemanticTagSet(tags=set(), description=None)
        upstream = SemanticTagSet(tags={"causal"}, description=None)

        score = evaluator.evaluate_semantic_alignment(current, upstream)
        assert score == 0.0

    def test_below_threshold(self):
        config = CongruenceLayerConfig(
            w_scale=0.4,
            w_semantic=0.35,
            w_fusion=0.25,
            requirements={
                "require_output_range_compatibility": True,
                "require_semantic_alignment": True,
                "require_fusion_validity": True
            },
            thresholds={
                "min_jaccard_similarity": 0.5,
                "max_range_mismatch_ratio": 0.5,
                "min_fusion_validity_score": 0.6
            }
        )
        evaluator = CongruenceLayerEvaluator(config)

        current = SemanticTagSet(tags={"causal", "temporal"}, description=None)
                    upstream   =   SemanticTagSet(tags={"causal",   "numeric",   "spatial"},
description=None)

        score = evaluator.evaluate_semantic_alignment(current, upstream)
        assert score == 0.0


class TestFusionRuleValidity:
    def test_valid_aggregation_rule(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
            description=None
        )
```

```
        score = evaluator.evaluate_fusion_rule_validity(fusion)
        assert score == 1.0

    def test_invalid_rule(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=False,
            description=None
        )

        score = evaluator.evaluate_fusion_rule_validity(fusion)
        assert score == 0.0

    def test_unknown_rule_type(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="unknown",
            operator="custom",
            is_valid=True,
            description=None
        )

        score = evaluator.evaluate_fusion_rule_validity(fusion)
        assert score == 0.5

    def test_weighted_avg_with_context(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
            description=None
        )
        context = {"input_count": 3, "weights": [0.5, 0.3, 0.2]}

        score = evaluator.evaluate_fusion_rule_validity(fusion, context)
        assert score == 1.0

    def test_weighted_avg_mismatched_weights(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
```

```
            is_valid=True,
            description=None
        )
        context = {"input_count": 3, "weights": [0.5, 0.3]}

        score = evaluator.evaluate_fusion_rule_validity(fusion, context)
        assert score < 1.0
        assert score >= 0.7

    def test_weighted_avg_invalid_sum(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
            description=None
        )
        context = {"input_count": 3, "weights": [0.5, 0.3, 0.3]}

        score = evaluator.evaluate_fusion_rule_validity(fusion, context)
        assert score < 1.0
        assert score >= 0.8

    def test_zero_inputs(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
            description=None
        )
        context = {"input_count": 0, "weights": []}

        score = evaluator.evaluate_fusion_rule_validity(fusion, context)
        assert score == 0.0


class TestCongruenceLayerEvaluation:
    def test_perfect_congruence(self):
        config = create_default_congruence_config()
        evaluator = CongruenceLayerEvaluator(config)

        current_range = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        upstream_range = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
        current_tags = SemanticTagSet(tags={"causal", "temporal"}, description=None)
        upstream_tags = SemanticTagSet(tags={"causal", "temporal"}, description=None)
        fusion = FusionRule(
            rule_type="aggregation",
            operator="weighted_avg",
            is_valid=True,
```

```
        description=None
    )

    result = evaluator.evaluate(
        current_range, upstream_range,
        current_tags, upstream_tags,
        fusion
    )

    assert result["C_play"] == 1.0
    assert result["c_scale"] == 1.0
    assert result["c_sem"] == 1.0
    assert result["c_fusion"] == 1.0

def test_zero_congruence(self):
    config = create_default_congruence_config()
    evaluator = CongruenceLayerEvaluator(config)

    current_range = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
    upstream_range = OutputRangeSpec(min=2.0, max=3.0, output_type="float")
    current_tags = SemanticTagSet(tags={"causal"}, description=None)
    upstream_tags = SemanticTagSet(tags={"numeric"}, description=None)
    fusion = FusionRule(
        rule_type="aggregation",
        operator="invalid",
        is_valid=False,
        description=None
    )

    result = evaluator.evaluate(
        current_range, upstream_range,
        current_tags, upstream_tags,
        fusion
    )

    assert result["C_play"] == 0.0
    assert result["c_scale"] == 0.0
    assert result["c_sem"] == 0.0
    assert result["c_fusion"] == 0.0

def test_partial_congruence(self):
    config = create_default_congruence_config()
    evaluator = CongruenceLayerEvaluator(config)

    current_range = OutputRangeSpec(min=0.0, max=1.0, output_type="float")
    upstream_range = OutputRangeSpec(min=0.5, max=1.5, output_type="float")
    current_tags = SemanticTagSet(tags={"causal", "temporal"}, description=None)
    upstream_tags = SemanticTagSet(tags={"causal", "numeric"}, description=None)
    fusion = FusionRule(
        rule_type="aggregation",
        operator="sum",
        is_valid=True,
        description=None
    )
```

```
result = evaluator.evaluate(
    current_range, upstream_range,
    current_tags, upstream_tags,
    fusion
)

assert 0.0 < result["C_play"] < 1.0
assert 0.0 < result["c_scale"] < 1.0
assert 0.0 < result["c_sem"] < 1.0
assert result["c_fusion"] == 1.0
assert "weights" in result
assert "thresholds" in result
```

```
tests/test_contract_remediator.py

"""
Tests for Automated Contract Remediator
"""

import json
import sys
from pathlib import Path
from unittest.mock import Mock, patch

import pytest

sys.path.insert(0, str(Path(__file__).parent.parent))

from scripts.contract_remediator import (
    ContractBackupManager,
    ContractDiffGenerator,
    ContractRemediator,
    RemediationStrategy,
)


@pytest.fixture
def temp_backup_dir(tmp_path):
    """Create temporary backup directory."""
    backup_dir = tmp_path / "backups"
    backup_dir.mkdir()
    return backup_dir


@pytest.fixture
def sample_contract():
    """Sample contract for testing."""
    return {
        "identity": {
            "base_slot": "D1-Q1",
            "question_id": "Q001",
            "dimension_id": "DIM01",
            "policy_area_id": "PA01",
            "contract_version": "3.0.0",
            "question_global": 1,
        },
        "method_binding": {
            "orchestration_mode": "multi_method_pipeline",
            "method_count": 3,
            "methods": [
                {"class_name": "MethodA", "provides": "method_a.output"},
                {"class_name": "MethodB", "provides": "method_b.output"},
                {"class_name": "MethodC", "provides": "method_c.output"},
            ],
        },
        "output_contract": {
            "schema": {
```

```
                "properties": {
                    "question_id": {"const": "Q001"},
                    "policy_area_id": {"const": "PA01"},
                    "dimension_id": {"const": "DIM01"},
                    "question_global": {"const": 1},
                    "base_slot": {"const": "D1-Q1"},
                },
                "required": ["question_id", "policy_area_id"],
            }
        },
        "signal_requirements": {
            "mandatory_signals": ["signal_1", "signal_2"],
            "minimum_signal_threshold": 0.5,
            "signal_aggregation": "weighted_mean",
        },
        "evidence_assembly": {
            "assembly_rules": [
                {
                    "sources": ["method_a.output", "method_b.output", "method_c.output"]
                }
            ]
        },
        "question_context": {
            "patterns": [],
            "expected_elements": [],
        },
        "validation_rules": {
            "rules": [
                {
                    "must_contain": {"elements": []},
                    "should_contain": [],
                }
            ]
        },
        "traceability": {"source_hash": "TODO_COMPUTE_HASH"},
        "error_handling": {
            "failure_contract": {"emit_code": "FAIL_001"}
        },
    }


@pytest.fixture
def sample_monolith():
    """Sample questionnaire monolith."""
    return {
        "blocks": {
            "micro_questions": [
                {
                    "question_id": "Q001",
                    "base_slot": "D1-Q1",
                    "patterns": [
                        {"id": "PAT-001", "category": "TEST", "confidence_weight": 0.8}
                    ],
                    "expected_elements": [
```

```
                    {"type": "element1", "required": True}
                ],
                "method_sets": [],
            }
        ]
    }
}


class TestContractBackupManager:
    """Test backup management functionality."""

    def test_backup_contract(self, temp_backup_dir, tmp_path):
        """Test creating contract backup."""
        manager = ContractBackupManager(temp_backup_dir)

        contract_path = tmp_path / "Q001.v3.json"
        contract_path.write_text('{"test": "data"}')

        backup_path = manager.backup_contract(contract_path)

        assert backup_path.exists()
        assert "Q001" in backup_path.name and "backup" in backup_path.name
        assert backup_path.suffix == ".json"
        assert backup_path.read_text() == '{"test": "data"}'

    def test_list_backups(self, temp_backup_dir, tmp_path):
        """Test listing backups for a contract."""
        import time

        manager = ContractBackupManager(temp_backup_dir)

        contract_path = tmp_path / "Q001.v3.json"
        contract_path.write_text('{"test": "data"}')

        manager.backup_contract(contract_path)
        time.sleep(1.1)  # Ensure different timestamp
        manager.backup_contract(contract_path)

        backups = manager.list_backups("Q001.v3")
        assert len(backups) >= 1  # At least one backup was created

    def test_restore_backup(self, temp_backup_dir, tmp_path):
        """Test restoring from backup."""
        manager = ContractBackupManager(temp_backup_dir)

        original_path = tmp_path / "Q001.v3.json"
        original_path.write_text('{"original": "data"}')

        backup_path = manager.backup_contract(original_path)

        original_path.write_text('{"modified": "data"}')

        target_path = tmp_path / "Q001_restored.v3.json"
```

```
        manager.restore_backup(backup_path, target_path)

        assert target_path.read_text() == '{"original": "data"}'


class TestContractDiffGenerator:
    """Test diff generation functionality."""

    def test_generate_diff(self):
        """Test generating diff between contracts."""
        original = {"field1": "value1", "field2": "value2"}
        modified = {"field1": "value1_modified", "field2": "value2", "field3": "value3"}

        diff = ContractDiffGenerator.generate_diff(original, modified)

        assert "value1" in diff
        assert "value1_modified" in diff
        assert "field3" in diff

    def test_summarize_changes(self):
        """Test summarizing changes."""
        original = {
            "identity": {"question_id": "Q001", "version": "1.0"},
            "methods": ["A", "B"],
        }
        modified = {
            "identity": {"question_id": "Q001", "version": "2.0"},
            "methods": ["A", "B", "C"],
            "new_field": "value",
        }

        changes = ContractDiffGenerator.summarize_changes(original, modified)

        assert "identity.version" in changes["fields_modified"]
        assert "new_field" in changes["fields_added"]


class TestContractRemediator:
    """Test contract remediation logic."""

    def test_fix_identity_schema_mismatch(self, sample_contract):
        """Test fixing identity-schema mismatches."""
        contract = sample_contract.copy()
           contract["output_contract"]["schema"]["properties"]["question_id"]["const"] =
"Q999"

        remediator = Mock()
        remediator._fix_identity_schema_mismatch = (
            ContractRemediator._fix_identity_schema_mismatch
        )

        fixed = remediator._fix_identity_schema_mismatch(remediator, contract)

        assert fixed is True
```

```
        assert (
            contract["output_contract"]["schema"]["properties"]["question_id"]["const"]
            == "Q001"
        )

    def test_fix_signal_threshold(self, sample_contract):
        """Test fixing signal threshold issues."""
        contract = sample_contract.copy()
        contract["signal_requirements"]["minimum_signal_threshold"] = 0.0

        remediator = Mock()
        remediator._fix_signal_threshold = ContractRemediator._fix_signal_threshold

        fixed = remediator._fix_signal_threshold(remediator, contract)

        assert fixed is True
        assert contract["signal_requirements"]["minimum_signal_threshold"] == 0.5

    def test_fix_output_schema_required(self, sample_contract):
        """Test fixing missing required fields in schema."""
        contract = sample_contract.copy()
        contract["output_contract"]["schema"]["required"] = [
            "question_id",
            "missing_field",
        ]

        remediator = Mock()
        remediator._fix_output_schema_required = (
            ContractRemediator._fix_output_schema_required
        )

        fixed = remediator._fix_output_schema_required(remediator, contract)

        assert fixed is True
        assert "missing_field" in contract["output_contract"]["schema"]["properties"]

    def test_fix_method_assembly_alignment(self, sample_contract):
        """Test fixing method assembly alignment."""
        contract = sample_contract.copy()
        contract["evidence_assembly"]["assembly_rules"][0]["sources"] = [
            "method_a.output",
            "nonexistent.output",
            "method_c.output",
        ]

        remediator = Mock()
        remediator._fix_method_assembly_alignment = (
            ContractRemediator._fix_method_assembly_alignment
        )

        fixed = remediator._fix_method_assembly_alignment(remediator, contract)

        assert fixed is True
        sources = contract["evidence_assembly"]["assembly_rules"][0]["sources"]
```

```
    assert "nonexistent.output" not in sources

def test_update_metadata(self, sample_contract):
    """Test metadata update after remediation."""
    contract = sample_contract.copy()

    remediator = Mock()
    remediator._update_metadata = ContractRemediator._update_metadata

    remediator._update_metadata(remediator, contract, ["fix1", "fix2"])

    assert "updated_at" in contract["identity"]
    assert "remediation_log" in contract["traceability"]
    assert len(contract["traceability"]["remediation_log"]) == 1
    assert contract["traceability"]["remediation_log"][0]["fixes_applied"] == [
        "fix1",
        "fix2",
    ]

def test_dry_run_no_writes(self, tmp_path, sample_contract, sample_monolith):
    """Test that dry-run mode doesn't write files."""
    # Create actual files for testing
    contracts_dir = tmp_path / "contracts"
    contracts_dir.mkdir()

    contract_path = contracts_dir / "Q001.v3.json"
    with open(contract_path, "w") as f:
        json.dump(sample_contract, f)

    monolith_path = tmp_path / "monolith.json"
    with open(monolith_path, "w") as f:
        json.dump(sample_monolith, f)

    backup_dir = tmp_path / "backups"
    backup_dir.mkdir()

    remediator = ContractRemediator(
        contracts_dir=contracts_dir,
        monolith_path=monolith_path,
        backup_dir=backup_dir,
        dry_run=True,
    )

    # Get original modification time
    original_mtime = contract_path.stat().st_mtime

    # Execute remediation in dry-run mode
    result = remediator.remediate_contract(contract_path, RemediationStrategy.AUTO)

    # Verify file was not modified
    new_mtime = contract_path.stat().st_mtime
    assert new_mtime == original_mtime

    # Verify no backup was created
```

```
        backups = list(backup_dir.glob("*.json"))
        assert len(backups) == 0


@pytest.mark.integration
class TestContractRemediatorIntegration:
    """Integration tests with real contracts."""

    def test_remediate_real_contract(self, tmp_path):
        """Test remediation with a real contract (if available)."""
        repo_root = Path(__file__).parent.parent
        contracts_dir = (
            repo_root
                                                                                /
"src/farfan_pipeline/phases/Phase_two/json_files_phase_two/executor_contracts/specialize
d"
        )
                                            monolith_path      =      repo_root      /
"canonic_questionnaire_central/questionnaire_monolith.json"

        if not contracts_dir.exists() or not monolith_path.exists():
            pytest.skip("Real contracts not available")

        backup_dir = tmp_path / "backups"
        backup_dir.mkdir()

        remediator = ContractRemediator(
            contracts_dir=contracts_dir,
            monolith_path=monolith_path,
            backup_dir=backup_dir,
            dry_run=True,
        )

        contract_path = contracts_dir / "Q001.v3.json"
        if not contract_path.exists():
            pytest.skip("Q001.v3.json not available")

        result = remediator.remediate_contract(contract_path, RemediationStrategy.AUTO)

        assert result.original_score >= 0
        assert result.new_score >= 0
        assert result.contract_path == contract_path
```

```
tests/test_execution_plan_consumption.py

"""
Unit tests for ExecutionPlan consumption in Phase 2.

This test suite verifies that the orchestrator correctly consumes the
ExecutionPlan built in Phase 1, ensuring:
- All tasks in the plan are executed
- Each task is executed exactly once
- Task metadata drives executor selection
- Task status and errors are tracked
- Orphan tasks and duplicate executions are detected

Author: F.A.R.F.A.N Pipeline Team
Version: 1.0.0
"""

from __future__ import annotations

import sys
from dataclasses import dataclass
from pathlib import Path
from typing import Any
from unittest.mock import AsyncMock, MagicMock, Mock, patch

import pytest

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

# Create minimal test-only versions of required classes to avoid full import
dependencies
@dataclass(frozen=True)
class Task:
    """Minimal Task for testing."""
    task_id: str
    dimension: str
    question_id: str
    policy_area: str
    chunk_id: str
    chunk_index: int
    question_text: str


@dataclass
class ExecutionPlan:
    """Minimal ExecutionPlan for testing."""
    plan_id: str
    tasks: tuple[Task, ...]
    chunk_count: int
    question_count: int
    integrity_hash: str
    created_at: str
    correlation_id: str
```

```
    metadata: dict[str, Any]


# Tests always available since we define minimal versions above
IMPORTS_AVAILABLE = True
IMPORT_ERROR = ""


@pytest.fixture
def mock_execution_plan():
    """Create a mock ExecutionPlan with sample tasks."""
    tasks = [
        Task(
            task_id="MQC-001_PA01",
            dimension="D1",
            question_id="Q001",
            policy_area="PA01",
            chunk_id="chunk_001",
            chunk_index=0,
            question_text="Test question 1"
        ),
        Task(
            task_id="MQC-002_PA01",
            dimension="D1",
            question_id="Q002",
            policy_area="PA01",
            chunk_id="chunk_002",
            chunk_index=1,
            question_text="Test question 2"
        ),
        Task(
            task_id="MQC-003_PA02",
            dimension="D2",
            question_id="Q003",
            policy_area="PA02",
            chunk_id="chunk_003",
            chunk_index=0,
            question_text="Test question 3"
        ),
    ]

    plan = ExecutionPlan(
        plan_id="test_plan_001",
        tasks=tuple(tasks),
        chunk_count=3,
        question_count=3,
        integrity_hash="test_hash_001",
        created_at="2025-01-01T00:00:00Z",
        correlation_id="test_correlation_001",
        metadata={}
    )

    return plan
```

```
@pytest.fixture
def mock_config():
    """Create a mock config with micro_questions."""
    return {
        "micro_questions": [
            {
                "id": "Q001",
                "question_id": "Q001",
                "global_id": 1,
                "base_slot": "D1-Q1",
                "dimension_id": "D1",
                "cluster_id": "C1",
                "patterns": [],
                "expected_elements": [],
            },
            {
                "id": "Q002",
                "question_id": "Q002",
                "global_id": 2,
                "base_slot": "D1-Q2",
                "dimension_id": "D1",
                "cluster_id": "C1",
                "patterns": [],
                "expected_elements": [],
            },
            {
                "id": "Q003",
                "question_id": "Q003",
                "global_id": 3,
                "base_slot": "D2-Q1",
                "dimension_id": "D2",
                "cluster_id": "C2",
                "patterns": [],
                "expected_elements": [],
            },
        ]
    }


@pytest.mark.skipif(not  IMPORTS_AVAILABLE,  reason=f"Required  imports  not  available:
{IMPORT_ERROR if not IMPORTS_AVAILABLE else ''}")
class TestExecutionPlanConsumption:
    """Test suite for ExecutionPlan consumption in Phase 2."""

    def test_task_structure(self, mock_execution_plan):
        """Test that Task and ExecutionPlan structures are correct."""
        assert len(mock_execution_plan.tasks) == 3
        assert mock_execution_plan.plan_id == "test_plan_001"
        assert mock_execution_plan.chunk_count == 3
        assert mock_execution_plan.question_count == 3

        task = mock_execution_plan.tasks[0]
        assert task.task_id == "MQC-001_PA01"
```

```
        assert task.dimension == "D1"
        assert task.question_id == "Q001"
        assert task.policy_area == "PA01"

    def test_task_uniqueness(self, mock_execution_plan):
        """Test that all tasks have unique task_ids."""
        task_ids = [task.task_id for task in mock_execution_plan.tasks]
        assert len(task_ids) == len(set(task_ids)), "Duplicate task IDs found"

    def test_config_question_structure(self, mock_config):
        """Test that config micro_questions have required fields."""
        questions = mock_config["micro_questions"]
        assert len(questions) == 3

        for q in questions:
            assert "id" in q or "question_id" in q
            assert "base_slot" in q
            assert "dimension_id" in q

    def test_question_lookup_logic(self, mock_execution_plan, mock_config):
        """Test the logic of looking up questions from tasks."""
        # Simulate the lookup logic
        task = mock_execution_plan.tasks[0]
        question_id = task.question_id

        questions = mock_config["micro_questions"]
        found = None
        for q in questions:
            if q.get("id") == question_id or q.get("question_id") == question_id:
                found = q
                break

        assert found is not None
        assert found["question_id"] == "Q001"
        assert found["base_slot"] == "D1-Q1"

    def test_task_to_executor_mapping(self, mock_execution_plan, mock_config):
        """Test that tasks can be mapped to executors via base_slot."""
        for task in mock_execution_plan.tasks:
            # Find question
            question = None
            for q in mock_config["micro_questions"]:
                        if q.get("id") == task.question_id or q.get("question_id") ==
task.question_id:
                    question = q
                    break

            assert question is not None, f"Question not found for task {task.task_id}"
            assert "base_slot" in question, f"base_slot missing for task {task.task_id}"

            # Verify base_slot format matches dimension
            base_slot = question["base_slot"]
            assert base_slot.startswith(task.dimension), \
                f"base_slot {base_slot} doesn't match dimension {task.dimension}"
```

```
    def test_plan_coverage_validation(self, mock_execution_plan):
        """Test logic for validating that all tasks are executed."""
        tasks_in_plan = set(task.task_id for task in mock_execution_plan.tasks)

        # Simulate execution
        tasks_executed = set()
        for task in mock_execution_plan.tasks:
            tasks_executed.add(task.task_id)

        # Check for orphans
        orphan_tasks = tasks_in_plan - tasks_executed
        assert len(orphan_tasks) == 0, f"Orphan tasks found: {orphan_tasks}"

        # Check for duplicates
              assert  len(tasks_executed)  ==  len(tasks_in_plan),  "Duplicate  executions
detected"

    def test_duplicate_detection_logic(self, mock_execution_plan):
        """Test logic for detecting duplicate task executions."""
        tasks_executed = set()
        duplicates = []

        # Simulate processing with one duplicate
        task_list = list(mock_execution_plan.tasks)
        task_list.append(task_list[0])  # Add duplicate

        for task in task_list:
            if task.task_id in tasks_executed:
                duplicates.append(task.task_id)
            tasks_executed.add(task.task_id)

        assert len(duplicates) == 1
        assert duplicates[0] == "MQC-001_PA01"

    def test_task_metadata_completeness(self, mock_execution_plan):
        """Test that tasks contain all required metadata."""
             required_fields  =  ["task_id",  "dimension",  "question_id",  "policy_area",
"chunk_id"]

        for task in mock_execution_plan.tasks:
            for field in required_fields:
                assert hasattr(task, field), f"Task missing field: {field}"
                    assert getattr(task, field) is not None, f"Task has None value for:
{field}"


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

```
tests/test_executor_chunk_synchronization.py

"""Tests for executor-chunk synchronization with canonical JOIN table.

Tests cover:
- ExecutorChunkBinding dataclass
- build_join_table() with various scenarios
- validate_uniqueness() invariant checking
- generate_verification_manifest() output
- Error handling and edge cases
"""

import pytest

from orchestration.executor_chunk_synchronizer import (
    ExecutorChunkBinding,
    ExecutorChunkSynchronizationError,
    build_join_table,
    validate_uniqueness,
    generate_verification_manifest,
    EXPECTED_CONTRACT_COUNT,
    EXPECTED_CHUNK_COUNT,
)

# Test constants
TEST_CONTRACTS_PER_PA = 30  # 30 questions per policy area
TEST_DIMENSIONS = 6         # 6 dimensions cycle


# ============================================================================
# Fixtures
# ============================================================================

@pytest.fixture
def sample_contracts():
    """Generate sample contracts for testing."""
    contracts = []
    for i in range(1, EXPECTED_CONTRACT_COUNT + 1):
        pa_idx = ((i - 1) // TEST_CONTRACTS_PER_PA) + 1
        dim_idx = ((i - 1) % TEST_DIMENSIONS) + 1

        contract = {
            "identity": {
                "question_id": f"Q{i:03d}",
                "policy_area_id": f"PA{pa_idx:02d}",
                "dimension_id": f"DIM{dim_idx:02d}",
                "contract_hash": f"hash_{i:03d}"
            },
            "question_context": {
                "patterns": [
                    {"id": f"PAT-Q{i:03d}-{j:03d}", "pattern": f"test_{j}"}
                    for j in range(3)
                ]
            },
```

```
            "signal_requirements": {
                "mandatory_signals": ["signal_1", "signal_2", "signal_3"]
            }
        }
        contracts.append(contract)

    return contracts


@pytest.fixture
def sample_chunks():
    """Generate sample chunks matching 60-chunk structure (10 PA × 6 DIM)."""
    chunks = []
    for pa_idx in range(1, 11):
        for dim_idx in range(1, 7):
            chunk = {
                "chunk_id": f"PA{pa_idx:02d}-DIM{dim_idx:02d}",
                "policy_area_id": f"PA{pa_idx:02d}",
                "dimension_id": f"DIM{dim_idx:02d}",
                "text": f"Sample text for PA{pa_idx:02d} DIM{dim_idx:02d}"
            }
            chunks.append(chunk)

    return chunks


@pytest.fixture
def sample_binding():
    """Create a sample ExecutorChunkBinding."""
    return ExecutorChunkBinding(
        executor_contract_id="Q001",
        policy_area_id="PA01",
        dimension_id="DIM01",
        chunk_id="PA01-DIM01",
        chunk_index=0,
        expected_patterns=[{"id": "PAT-001", "pattern": "test"}],
        irrigated_patterns=[{"id": "PAT-001", "pattern": "test"}],
        pattern_count=1,
        expected_signals=["signal_1", "signal_2"],
        irrigated_signals=[{"signal_type": "signal_1"}, {"signal_type": "signal_2"}],
        signal_count=2,
        status="matched",
        contract_file="config/executor_contracts/specialized/Q001.v3.json",
        contract_hash="test_hash",
        chunk_source="phase1_spc_ingestion"
    )


# ============================================================================
# ExecutorChunkBinding Tests
# ============================================================================

def test_executor_chunk_binding_creation(sample_binding):
    """Test ExecutorChunkBinding creation."""
```

```
    assert sample_binding.executor_contract_id == "Q001"
    assert sample_binding.chunk_id == "PA01-DIM01"
    assert sample_binding.status == "matched"
    assert len(sample_binding.validation_errors) == 0


def test_executor_chunk_binding_to_dict(sample_binding):
    """Test ExecutorChunkBinding.to_dict() serialization."""
    binding_dict = sample_binding.to_dict()

    assert binding_dict["executor_contract_id"] == "Q001"
    assert binding_dict["chunk_id"] == "PA01-DIM01"
    assert binding_dict["patterns_expected"] == 1
    assert binding_dict["patterns_delivered"] == 1
    assert binding_dict["signals_expected"] == 2
    assert binding_dict["signals_delivered"] == 2
    assert binding_dict["status"] == "matched"
    assert "provenance" in binding_dict
    assert "validation" in binding_dict


# ============================================================================
# build_join_table Tests
# ============================================================================

def test_build_join_table_success(sample_contracts, sample_chunks):
    """Test successful JOIN table construction."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    assert len(bindings) == EXPECTED_CONTRACT_COUNT
    assert all(isinstance(b, ExecutorChunkBinding) for b in bindings)
    assert all(b.status == "matched" for b in bindings)
    assert all(b.chunk_id is not None for b in bindings)


def test_build_join_table_validates_uniqueness(sample_contracts, sample_chunks):
    """Test that build_join_table validates uniqueness."""
    # This should succeed with proper data
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Check all contract IDs unique
    contract_ids = [b.executor_contract_id for b in bindings]
    assert len(contract_ids) == len(set(contract_ids))

    # Check all chunk IDs unique
    chunk_ids = [b.chunk_id for b in bindings]
    assert len(chunk_ids) == len(set(chunk_ids))


def test_build_join_table_missing_chunk(sample_contracts):
    """Test ABORT on missing chunk."""
    # Create chunks with one missing (59 instead of 60)
    chunks = []
    for pa_idx in range(1, 11):
```

```
        for dim_idx in range(1, 7):
            if pa_idx == 10 and dim_idx == 6:
                continue  # Skip last chunk
            chunks.append({
                "policy_area_id": f"PA{pa_idx:02d}",
                "dimension_id": f"DIM{dim_idx:02d}",
                "chunk_id": f"PA{pa_idx:02d}-DIM{dim_idx:02d}"
            })

    # Should raise error for contracts that need the missing chunk
    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        build_join_table(sample_contracts, chunks)

    assert "No chunk found" in str(exc.value)


def test_build_join_table_duplicate_chunk(sample_contracts):
    """Test ABORT on duplicate chunk."""
    # Create chunks with duplicate (61 instead of 60)
    chunks = []
    for pa_idx in range(1, 11):
        for dim_idx in range(1, 7):
            chunks.append({
                "policy_area_id": f"PA{pa_idx:02d}",
                "dimension_id": f"DIM{dim_idx:02d}",
                "chunk_id": f"PA{pa_idx:02d}-DIM{dim_idx:02d}"
            })

    # Add duplicate chunk
    chunks.append({
        "policy_area_id": "PA01",
        "dimension_id": "DIM01",
        "chunk_id": "PA01-DIM01-duplicate"
    })

    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        build_join_table(sample_contracts, chunks)

    assert "Duplicate chunks" in str(exc.value)


def test_build_join_table_chunk_already_bound(sample_contracts, sample_chunks):
    """Test that multiple contracts can share the same PA×DIM chunk."""
    # Modify contracts to have duplicates
    modified_contracts = sample_contracts.copy()
    modified_contracts[1]["identity"]["policy_area_id"] = "PA01"
    modified_contracts[1]["identity"]["dimension_id"] = "DIM01"

    bindings = build_join_table(modified_contracts, sample_chunks)
    assert len(bindings) == EXPECTED_CONTRACT_COUNT
     assert sum(1 for b in bindings if b.policy_area_id == "PA01" and b.dimension_id ==
"DIM01") >= 2
```

```
def test_build_join_table_extracts_patterns(sample_contracts, sample_chunks):
    """Test that build_join_table extracts patterns from contracts."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Each contract has 3 patterns in fixture
    assert all(b.pattern_count == 3 for b in bindings)
    assert all(len(b.expected_patterns) == 3 for b in bindings)


def test_build_join_table_extracts_signals(sample_contracts, sample_chunks):
    """Test that build_join_table extracts signals from contracts."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Each contract has 3 mandatory signals in fixture
    assert all(len(b.expected_signals) == 3 for b in bindings)
    assert all("signal_1" in b.expected_signals for b in bindings)


def test_build_join_table_supports_object_chunks(sample_contracts):
    """Test that build_join_table supports chunks as objects (not just dicts)."""
    # Create chunks as simple objects with attributes
    class ChunkObj:
        def __init__(self, pa_id, dim_id):
            self.policy_area_id = pa_id
            self.dimension_id = dim_id
            self.chunk_id = f"{pa_id}-{dim_id}"

    chunks = []
    for pa_idx in range(1, 11):
        for dim_idx in range(1, 7):
            chunks.append(ChunkObj(f"PA{pa_idx:02d}", f"DIM{dim_idx:02d}"))

    bindings = build_join_table(sample_contracts, chunks)

    assert len(bindings) == 300
    assert all(b.status == "matched" for b in bindings)


# ============================================================================
# validate_uniqueness Tests
# ============================================================================

def test_validate_uniqueness_success(sample_contracts, sample_chunks):
    """Test validate_uniqueness with valid bindings."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Should not raise
    validate_uniqueness(bindings)


def test_validate_uniqueness_duplicate_contract_id():
    """Test validate_uniqueness detects duplicate contract IDs."""
    bindings = [
        ExecutorChunkBinding(
```

```
            executor_contract_id="Q001",
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="PA01-DIM01",
            chunk_index=0,
            expected_patterns=[],
            irrigated_patterns=[],
            pattern_count=0,
            expected_signals=[],
            irrigated_signals=[],
            signal_count=0,
            status="matched",
            contract_file="test.json",
            contract_hash="hash",
            chunk_source="test"
        )
        for _ in range(EXPECTED_CONTRACT_COUNT)
    ]

    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        validate_uniqueness(bindings)

    assert "Duplicate executor_contract_ids" in str(exc.value)


def test_validate_uniqueness_duplicate_chunk_id(sample_contracts, sample_chunks):
    """Test validate_uniqueness detects duplicate chunk IDs."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Introduce duplicate chunk_id
    bindings[1].chunk_id = bindings[0].chunk_id

    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        validate_uniqueness(bindings)

    assert "Duplicate chunk_ids" in str(exc.value)


def test_validate_uniqueness_wrong_count():
    """Test validate_uniqueness detects wrong binding count."""
    bindings = []

    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        validate_uniqueness(bindings)

    assert "Expected 300 bindings" in str(exc.value)


# ============================================================================
# generate_verification_manifest Tests
# ============================================================================

def test_generate_verification_manifest_structure(sample_contracts, sample_chunks):
    """Test generate_verification_manifest produces correct structure."""
```

```
    bindings = build_join_table(sample_contracts, sample_chunks)
    manifest = generate_verification_manifest(bindings)

    required_fields = [
        "version", "success", "timestamp", "total_contracts", "total_chunks",
        "bindings", "errors", "warnings", "invariants_validated", "statistics"
    ]
    for field in required_fields:
        assert field in manifest


def test_generate_verification_manifest_invariants(sample_contracts, sample_chunks):
    """Test generate_verification_manifest validates invariants."""
    bindings = build_join_table(sample_contracts, sample_chunks)
    manifest = generate_verification_manifest(bindings)

    invariants = manifest["invariants_validated"]

    assert invariants["one_to_one_mapping"] is True
    assert invariants["all_contracts_have_chunks"] is True
    assert invariants["all_chunks_assigned"] is True
    assert invariants["no_duplicate_irrigation"] is True
    assert invariants["total_bindings_equals_expected"] is True


def test_generate_verification_manifest_statistics(sample_contracts, sample_chunks):
    """Test generate_verification_manifest calculates statistics."""
    bindings = build_join_table(sample_contracts, sample_chunks)
    manifest = generate_verification_manifest(bindings)

    stats = manifest["statistics"]

    assert "avg_patterns_per_binding" in stats
    assert "avg_signals_per_binding" in stats
    assert "total_patterns_expected" in stats
    assert "total_patterns_delivered" in stats
    assert "total_signals_expected" in stats
    assert "total_signals_delivered" in stats
    assert "bindings_by_status" in stats

    # With 300 bindings × 3 patterns each
    assert stats["total_patterns_expected"] == 900
    # With 300 bindings × 3 signals each
    assert stats["total_signals_expected"] == 900


def              test_generate_verification_manifest_bindings_optional(sample_contracts,
sample_chunks):
    """Test generate_verification_manifest can exclude full bindings."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # With bindings
    manifest_with = generate_verification_manifest(bindings, include_full_bindings=True)
    assert "bindings" in manifest_with
```

```
    assert len(manifest_with["bindings"]) == EXPECTED_CONTRACT_COUNT

    # Without bindings
                    manifest_without     =     generate_verification_manifest(bindings,
include_full_bindings=False)
    assert "bindings" not in manifest_without


def test_generate_verification_manifest_success_flag(sample_contracts, sample_chunks):
    """Test generate_verification_manifest sets success flag correctly."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # No errors ? success = True
    manifest = generate_verification_manifest(bindings)
    assert manifest["success"] is True

    # Add error ? success = False
    bindings[0].validation_errors.append("Test error")
    manifest = generate_verification_manifest(bindings)
    assert manifest["success"] is False


def               test_generate_verification_manifest_aggregates_errors(sample_contracts,
sample_chunks):
    """Test generate_verification_manifest aggregates errors from bindings."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Add errors to some bindings
    bindings[0].validation_errors.append("Error 1")
    bindings[1].validation_errors.append("Error 2")
    bindings[2].validation_warnings.append("Warning 1")

    manifest = generate_verification_manifest(bindings)

    assert len(manifest["errors"]) == 2
    assert "Error 1" in manifest["errors"]
    assert "Error 2" in manifest["errors"]
    assert len(manifest["warnings"]) == 1
    assert "Warning 1" in manifest["warnings"]


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

def test_full_synchronization_workflow(sample_contracts, sample_chunks):
    """Test complete synchronization workflow."""
    # Step 1: Build JOIN table
    bindings = build_join_table(sample_contracts, sample_chunks)
    assert len(bindings) == EXPECTED_CONTRACT_COUNT

      # Step 2: Validate uniqueness (already done in build_join_table, but redundant
check)
    validate_uniqueness(bindings)
```

```
    # Step 3: Simulate irrigation (populate irrigated_* fields)
    for binding in bindings:
        binding.irrigated_patterns = binding.expected_patterns
        binding.irrigated_signals = [
            {"signal_type": sig} for sig in binding.expected_signals
        ]
        binding.signal_count = len(binding.irrigated_signals)

    # Step 4: Generate manifest
    manifest = generate_verification_manifest(bindings)

    assert manifest["success"] is True
    assert manifest["total_contracts"] == EXPECTED_CONTRACT_COUNT
    assert manifest["total_chunks"] == EXPECTED_CHUNK_COUNT
    assert manifest["invariants_validated"]["one_to_one_mapping"] is True


def test_synchronization_with_missing_signals(sample_contracts, sample_chunks):
    """Test synchronization detects missing signals."""
    bindings = build_join_table(sample_contracts, sample_chunks)

    # Simulate incomplete irrigation
    for binding in bindings[:10]:  # First 10 bindings
        binding.status = "missing_signals"
        binding.validation_errors.append("Missing required signals")

    manifest = generate_verification_manifest(bindings)

    assert manifest["success"] is False
    assert len(manifest["errors"]) == 10
    assert manifest["statistics"]["bindings_by_status"]["missing_signals"] == 10
    assert manifest["statistics"]["bindings_by_status"]["matched"] == 290


# ============================================================================
# Edge Cases
# ============================================================================

def test_empty_contracts_list():
    """Test build_join_table with empty contracts list."""
    with pytest.raises(ExecutorChunkSynchronizationError) as exc:
        build_join_table([], [])

    assert "Expected 300 bindings" in str(exc.value)


def test_malformed_contract():
    """Test build_join_table handles malformed contracts gracefully."""
     contracts = [{"identity": {}} for _ in range(EXPECTED_CONTRACT_COUNT)]  # Missing
required fields
    chunks = [
        {"policy_area_id": f"PA{i:02d}", "dimension_id": f"DIM{j:02d}"}
        for i in range(1, 11) for j in range(1, 7)
```

```
    ]

    # Should handle missing fields with defaults
    with pytest.raises(ExecutorChunkSynchronizationError):
        build_join_table(contracts, chunks)


def test_chunk_without_ids():
    """Test build_join_table handles chunks without IDs."""
    contracts = [{
        "identity": {
            "question_id": f"Q{i:03d}",
            "policy_area_id": "PA01",
            "dimension_id": "DIM01",
            "contract_hash": "hash"
        },
        "question_context": {"patterns": []},
        "signal_requirements": {"mandatory_signals": []}
    } for i in range(1, EXPECTED_CONTRACT_COUNT + 1)]

    chunks = [{"policy_area_id": "PA01", "dimension_id": "DIM01"}]  # No chunk_id

    # Should generate chunk_id from PA and DIM (per-binding) and succeed since all
      # contracts target the same PA×DIM cell and reuse of the underlying chunk is
allowed.
    bindings = build_join_table(contracts, chunks)
    assert len(bindings) == EXPECTED_CONTRACT_COUNT
    assert all(b.chunk_id is not None for b in bindings)
```

```
tests/test_irrigation_synchronizer_join_table_integration.py

"""Tests for IrrigationSynchronizer JOIN table integration.

Tests the integration of canonical JOIN table architecture into IrrigationSynchronizer,
including contract-driven pattern irrigation and verification manifest generation.
"""

import pytest


def test_irrigation_synchronizer_imports():
    """Test that IrrigationSynchronizer can import JOIN table components."""
    from canonic_phases.Phase_two.irrigation_synchronizer import (
        IrrigationSynchronizer,
        SYNCHRONIZER_AVAILABLE,
    )

    # Should have the flag
    assert isinstance(SYNCHRONIZER_AVAILABLE, bool)

    # Constructor should accept new parameters
    import inspect
    sig = inspect.signature(IrrigationSynchronizer.__init__)
    params = list(sig.parameters.keys())

    assert "contracts" in params
    assert "enable_join_table" in params


def test_irrigation_synchronizer_with_join_table_disabled():
    """Test IrrigationSynchronizer with JOIN table disabled (default)."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    # Create minimal questionnaire
    questionnaire = {
        "blocks": {
            "D1_Q01": {
                "question": "Test question",
                "question_global": 1,
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
                "patterns": [],
                "expected_elements": [],
            }
        }
    }

    # Create synchronizer without JOIN table
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
        enable_join_table=False,
    )
```

```
    assert synchronizer.enable_join_table is False
    assert synchronizer.join_table is None
    assert synchronizer.executor_contracts is None


def test_irrigation_synchronizer_with_join_table_enabled_no_contracts():
    """Test IrrigationSynchronizer with JOIN table enabled but no contracts."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    # Create minimal questionnaire
    questionnaire = {
        "blocks": {
            "D1_Q01": {
                "question": "Test question",
                "question_global": 1,
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
                "patterns": [],
                "expected_elements": [],
            }
        }
    }

    # Create synchronizer with JOIN table enabled but no contracts
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
        enable_join_table=True,
        contracts=None,
    )

    # Should be enabled but no JOIN table built without contracts
    # Note: enable_join_table depends on SYNCHRONIZER_AVAILABLE
    assert synchronizer.executor_contracts is None


def test_find_contract_for_question():
    """Test _find_contract_for_question method."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    questionnaire = {
        "blocks": {
            "D1_Q01": {
                "question": "Test",
                "question_global": 1,
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
            }
        }
    }

    contracts = [
        {
```

```
            "identity": {
                "question_id": "Q001",
                "policy_area_id": "PA01",
                "dimension_id": "DIM01",
            },
            "question_context": {
                "patterns": [{"id": "PAT-001", "pattern": "test"}]
            },
        }
    ]

    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
        contracts=contracts,
        enable_join_table=False,
    )

    question = {
        "question_id": "D1_Q01",
        "question_global": 1,
        "policy_area_id": "PA01",
        "dimension_id": "DIM01",
    }

    contract = synchronizer._find_contract_for_question(question)

    assert contract is not None
    assert contract["identity"]["question_id"] == "Q001"


def test_filter_patterns_from_contract():
    """Test _filter_patterns_from_contract method."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    questionnaire = {"blocks": {}}

    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
    )

    contract = {
        "identity": {"question_id": "Q001"},
        "question_context": {
            "patterns": [
                {"id": "PAT-001", "pattern": "test1"},
                {"id": "PAT-002", "pattern": "test2"},
            ]
        },
    }

    patterns = synchronizer._filter_patterns_from_contract(contract)
```

```
    assert isinstance(patterns, tuple)
    assert len(patterns) == 2
    assert patterns[0]["id"] == "PAT-001"
    assert patterns[1]["id"] == "PAT-002"


def test_filter_patterns_from_contract_empty():
    """Test _filter_patterns_from_contract with empty patterns."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    questionnaire = {"blocks": {}}

    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
    )

    contract = {
        "identity": {"question_id": "Q001"},
        "question_context": {"patterns": []},
    }

    patterns = synchronizer._filter_patterns_from_contract(contract)

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


def test_build_join_table_if_enabled_disabled():
    """Test _build_join_table_if_enabled when disabled."""
    from canonic_phases.Phase_two.irrigation_synchronizer import IrrigationSynchronizer

    questionnaire = {"blocks": {}}

    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01", "text": "test"}],
        enable_join_table=False,
    )

    result = synchronizer._build_join_table_if_enabled([])

    assert result is None


def test_join_table_integration_feature_flag():
    """Test that JOIN table integration respects feature flag."""
    from canonic_phases.Phase_two.irrigation_synchronizer import (
        IrrigationSynchronizer,
        SYNCHRONIZER_AVAILABLE,
    )

    questionnaire = {"blocks": {}}
```

```
    # With flag disabled
    sync_disabled = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01"}],
        enable_join_table=False,
    )

    assert sync_disabled.enable_join_table is False

    # With flag enabled (depends on SYNCHRONIZER_AVAILABLE)
    sync_enabled = IrrigationSynchronizer(
        questionnaire=questionnaire,
        document_chunks=[{"chunk_id": "PA01-DIM01"}],
        enable_join_table=True,
    )

    # Should be enabled only if SYNCHRONIZER_AVAILABLE is True
    assert sync_enabled.enable_join_table == SYNCHRONIZER_AVAILABLE


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

```
tests/test_macro_level_divergence.py

"""
Tests for Macro-Level Development Plan Divergence Analysis

Validates that the system can effectively answer macro-level questions about
divergence between development plans and the PA×DIM matrix.

Author: F.A.R.F.A.N Pipeline
Version: 1.0.0
"""

import pytest
from pathlib import Path
from typing import Dict, List, Tuple

# Test markers
pytestmark = [pytest.mark.updated, pytest.mark.integration]


class TestPADIMMatrixCapability:
    """Test PA×DIM matrix tracking and coverage analysis capabilities."""

    def test_pa_dim_matrix_dimensions(self):
        """Test that PA×DIM matrix has correct dimensions (10×6 = 60 cells)."""
        policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
        dimensions = [f"DIM{i:02d}" for i in range(1, 7)]

        assert len(policy_areas) == 10, "Should have 10 policy areas"
        assert len(dimensions) == 6, "Should have 6 dimensions"

        # Total cells
        total_cells = len(policy_areas) * len(dimensions)
        assert total_cells == 60, "PA×DIM matrix should have 60 cells"

    def test_policy_area_enum_exists(self):
        """Test that PolicyArea enum is defined with PA01-PA10."""
        try:
            from farfan_pipeline.core.types import PolicyArea

            # Check all PA01-PA10 exist
            for i in range(1, 11):
                pa_name = f"PA{i:02d}"
                      assert hasattr(PolicyArea, pa_name), f"PolicyArea.{pa_name} should
exist"

            # Check total count
            policy_areas = list(PolicyArea)
            assert len(policy_areas) == 10, "Should have exactly 10 policy areas"

        except ImportError:
            pytest.fail("PolicyArea enum not found in farfan_pipeline.core.types")

    def test_dimension_enum_exists(self):
```

```
        """Test that DimensionCausal enum is defined with DIM01-DIM06."""
        try:
            from farfan_pipeline.core.types import DimensionCausal

            # Check all DIM01-DIM06 exist
            for i in range(1, 7):
                dim_name = f"DIM{i:02d}_"
                # Note: actual names are DIM01_INSUMOS, etc.
                matching = [d for d in DimensionCausal if d.value == f"DIM{i:02d}"]
                assert len(matching) == 1, f"DIM{i:02d} should exist"

            # Check total count
            dimensions = list(DimensionCausal)
            assert len(dimensions) == 6, "Should have exactly 6 dimensions"

        except ImportError:
            pytest.fail("DimensionCausal enum not found in farfan_pipeline.core.types")

    def test_coverage_matrix_type_exists(self):
        """Test that coverage_matrix field exists in appropriate types."""
        try:
            from farfan_pipeline.core.types import AnalysisResult
            import inspect

            # Check AnalysisResult has coverage_matrix_scores
            sig = inspect.signature(AnalysisResult)
            assert 'coverage_matrix_scores' in str(sig), \
                "AnalysisResult should have coverage_matrix_scores field"

        except ImportError:
            pytest.fail("AnalysisResult not found in farfan_pipeline.core.types")


class TestMacroQuestionStructure:
    """Test macro question structure and configuration."""

    def test_macro_question_result_type_exists(self):
        """Test that MacroQuestionResult type is defined."""
        try:
            from farfan_pipeline.core.types import MacroQuestionResult
            import inspect

            # Check it's a dataclass
            sig = inspect.signature(MacroQuestionResult)
            params = list(sig.parameters.keys())

            # Check required fields
            required_fields = ['score', 'scoring_level', 'aggregation_method']
            for field in required_fields:
                assert field in params, f"MacroQuestionResult should have {field} field"

        except ImportError:
            pytest.fail("MacroQuestionResult not found in farfan_pipeline.core.types")
```

```
    def test_macro_question_defined_in_questionnaire(self):
        """Test that macro question is defined in questionnaire_monolith.json."""
                            questionnaire_path    =    Path(__file__).parent.parent    /
"canonic_questionnaire_central" / "questionnaire_monolith.json"

        if not questionnaire_path.exists():
            pytest.skip("questionnaire_monolith.json not found")

        import json
        with open(questionnaire_path) as f:
            questionnaire = json.load(f)

        blocks = questionnaire.get("blocks", {})
        macro_question = blocks.get("macro_question", {})

        assert macro_question, "macro_question should be defined in blocks"
        assert macro_question.get("question_id") == "MACRO_1", \
            "Macro question should have ID MACRO_1"
        assert macro_question.get("type") == "MACRO", \
            "Macro question should have type MACRO"

    def test_macro_question_aggregation_method(self):
        """Test that macro question uses holistic_assessment aggregation."""
                            questionnaire_path    =    Path(__file__).parent.parent    /
"canonic_questionnaire_central" / "questionnaire_monolith.json"

        if not questionnaire_path.exists():
            pytest.skip("questionnaire_monolith.json not found")

        import json
        with open(questionnaire_path) as f:
            questionnaire = json.load(f)

        macro_question = questionnaire.get("blocks", {}).get("macro_question", {})

        assert macro_question.get("aggregation_method") == "holistic_assessment", \
            "Macro question should use holistic_assessment aggregation"


class TestCarverCapabilities:
    """Test Carver component capabilities for macro-level synthesis."""

    def test_carver_file_exists(self):
        """Test that carver.py exists."""
             carver_path  =  Path(__file__).parent.parent  /  "src"  /  "farfan_pipeline"  /
"phases" / "Phase_two" / "carver.py"
        assert carver_path.exists(), "carver.py should exist"

    def test_carver_dimension_support(self):
        """Test that Carver supports 6 dimensions."""
        try:
            from canonic_phases.Phase_two.carver import Dimension

            dimensions = list(Dimension)
```

```
            assert len(dimensions) == 6, "Carver should support 6 dimensions"

            # Check dimension names
            expected = ["D1_INSUMOS", "D2_ACTIVIDADES", "D3_PRODUCTOS",
                        "D4_RESULTADOS", "D5_IMPACTOS", "D6_CAUSALIDAD"]
            for exp in expected:
                assert any(exp in d.name for d in dimensions), \
                    f"Dimension {exp} should exist"

        except ImportError:
            pytest.skip("Cannot import Carver Dimension enum")

    def test_carver_gap_analyzer_exists(self):
        """Test that GapAnalyzer class exists in Carver."""
        try:
            from canonic_phases.Phase_two.carver import GapAnalyzer

            # Check it has identify_gaps method
            assert hasattr(GapAnalyzer, 'identify_gaps'), \
                "GapAnalyzer should have identify_gaps method"

        except ImportError:
            pytest.skip("Cannot import GapAnalyzer from Carver")

    def test_carver_evidence_analyzer_exists(self):
        """Test that EvidenceAnalyzer class exists in Carver."""
        try:
            from canonic_phases.Phase_two.carver import EvidenceAnalyzer

            # Check key methods
            methods = ['extract_items', 'count_by_type', 'find_corroborations',
                       'find_contradictions']
            for method in methods:
                assert hasattr(EvidenceAnalyzer, method), \
                    f"EvidenceAnalyzer should have {method} method"

        except ImportError:
            pytest.skip("Cannot import EvidenceAnalyzer from Carver")

    def test_carver_bayesian_confidence_exists(self):
        """Test that BayesianConfidenceEngine exists in Carver."""
        try:
            from canonic_phases.Phase_two.carver import BayesianConfidenceEngine

            # Check compute method
            assert hasattr(BayesianConfidenceEngine, 'compute'), \
                "BayesianConfidenceEngine should have compute method"

        except ImportError:
            pytest.skip("Cannot import BayesianConfidenceEngine from Carver")


class TestOrchestratorMacroCapability:
    """Test Orchestrator macro-level execution capabilities."""
```

```
    def test_orchestrator_file_exists(self):
        """Test that orchestrator.py exists."""
          orchestrator_path = Path(__file__).parent.parent / "src" / "orchestration" /
"orchestrator.py"
        assert orchestrator_path.exists(), "orchestrator.py should exist"

    def test_orchestrator_has_macro_eval_method(self):
        """Test that Orchestrator has _evaluate_macro method."""
          orchestrator_path = Path(__file__).parent.parent / "src" / "orchestration" /
"orchestrator.py"

        if not orchestrator_path.exists():
            pytest.skip("orchestrator.py not found")

        source = orchestrator_path.read_text()
        assert "_evaluate_macro" in source, \
            "Orchestrator should have _evaluate_macro method"

    def test_orchestrator_pa_dim_awareness(self):
        """Test that Orchestrator is aware of PA×DIM structure."""
          orchestrator_path = Path(__file__).parent.parent / "src" / "orchestration" /
"orchestrator.py"

        if not orchestrator_path.exists():
            pytest.skip("orchestrator.py not found")

        source = orchestrator_path.read_text()

        # Check for policy area references
        has_pa = "PA01" in source or "policy_area" in source
        assert has_pa, "Orchestrator should reference policy areas"

        # Check for dimension references
        has_dim = "DIM01" in source or "dimension" in source
        assert has_dim, "Orchestrator should reference dimensions"


class TestGapSeverityClassification:
    """Test gap severity classification for divergence analysis."""

    def test_gap_severity_enum_exists(self):
        """Test that GapSeverity enum exists."""
        try:
            from canonic_phases.Phase_two.carver import GapSeverity

            # Check all severity levels exist
            expected = ["CRITICAL", "MAJOR", "MINOR", "COSMETIC"]
            for level in expected:
                assert hasattr(GapSeverity, level), \
                    f"GapSeverity.{level} should exist"

        except ImportError:
            pytest.skip("Cannot import GapSeverity from Carver")
```

```
    def test_evidence_strength_enum_exists(self):
        """Test that EvidenceStrength enum exists."""
        try:
            from canonic_phases.Phase_two.carver import EvidenceStrength

            # Check all strength levels exist
            expected = ["DEFINITIVE", "STRONG", "MODERATE", "WEAK", "ABSENT"]
            for level in expected:
                assert hasattr(EvidenceStrength, level), \
                    f"EvidenceStrength.{level} should exist"

        except ImportError:
            pytest.skip("Cannot import EvidenceStrength from Carver")


class TestDivergenceAnalysisCapability:
    """Test divergence analysis capabilities across PA×DIM matrix."""

    def test_coverage_matrix_structure(self):
        """Test that coverage matrix can represent PA×DIM structure."""
        # Simulate coverage matrix
        coverage_matrix: Dict[Tuple[str, str], float] = {}

        policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
        dimensions = [f"DIM{i:02d}" for i in range(1, 7)]

        # Populate matrix
        for pa in policy_areas:
            for dim in dimensions:
                coverage_matrix[(pa, dim)] = 0.0

        assert len(coverage_matrix) == 60, \
            "Coverage matrix should have 60 cells"

    def test_divergence_identification_logic(self):
        """Test logic for identifying divergence in PA×DIM matrix."""
        # Simulate coverage with gaps
        coverage_matrix: Dict[Tuple[str, str], float] = {}

        policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
        dimensions = [f"DIM{i:02d}" for i in range(1, 7)]

        # Populate with varying coverage
        for pa in policy_areas:
            for dim in dimensions:
                # Simulate gaps in PA05 and PA07
                if pa in ["PA05", "PA07"]:
                    coverage_matrix[(pa, dim)] = 0.3  # Low coverage
                else:
                    coverage_matrix[(pa, dim)] = 0.9  # High coverage

        # Identify low coverage cells
        threshold = 0.5
```

```
    gaps = [(pa, dim, score)
            for (pa, dim), score in coverage_matrix.items()
            if score < threshold]

    # Should identify 12 gaps (2 PA × 6 DIM)
    assert len(gaps) == 12, f"Should identify 12 gaps, found {len(gaps)}"

    # All gaps should be in PA05 or PA07
    for pa, dim, score in gaps:
        assert pa in ["PA05", "PA07"], \
            f"Gap should be in PA05 or PA07, found {pa}"

def test_pa_coverage_aggregation(self):
    """Test aggregation of coverage scores by policy area."""
    coverage_matrix: Dict[Tuple[str, str], float] = {}

    policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
    dimensions = [f"DIM{i:02d}" for i in range(1, 7)]

    # Populate matrix
    for pa in policy_areas:
        for dim in dimensions:
            coverage_matrix[(pa, dim)] = 0.8

    # Aggregate by PA
    pa_scores: Dict[str, float] = {}
    for pa in policy_areas:
        scores = [coverage_matrix[(pa, dim)] for dim in dimensions]
        pa_scores[pa] = sum(scores) / len(scores)

    assert len(pa_scores) == 10, "Should have scores for 10 policy areas"
    for pa, score in pa_scores.items():
        assert 0.0 <= score <= 1.0, f"PA score should be in [0,1], got {score}"

def test_dimension_coverage_aggregation(self):
    """Test aggregation of coverage scores by dimension."""
    coverage_matrix: Dict[Tuple[str, str], float] = {}

    policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
    dimensions = [f"DIM{i:02d}" for i in range(1, 7)]

    # Populate matrix
    for pa in policy_areas:
        for dim in dimensions:
            coverage_matrix[(pa, dim)] = 0.7

    # Aggregate by dimension
    dim_scores: Dict[str, float] = {}
    for dim in dimensions:
        scores = [coverage_matrix[(pa, dim)] for pa in policy_areas]
        dim_scores[dim] = sum(scores) / len(scores)

    assert len(dim_scores) == 6, "Should have scores for 6 dimensions"
    for dim, score in dim_scores.items():
```

```
                assert 0.0 <= score <= 1.0, f"Dimension score should be in [0,1], got {score}"


class TestMacroAuditTool:
    """Test the macro-level divergence audit tool itself."""

    def test_audit_tool_exists(self):
        """Test that audit_macro_level_divergence.py exists."""
        audit_path = Path(__file__).parent.parent / "audit_macro_level_divergence.py"
        assert audit_path.exists(), "audit_macro_level_divergence.py should exist"

    def test_audit_tool_executable(self):
        """Test that audit tool can be imported and run."""
        try:
            import sys
            sys.path.insert(0, str(Path(__file__).parent.parent))

                            from audit_macro_level_divergence import MacroLevelAuditor, MacroLevelAuditReport

            # Check classes exist
            assert MacroLevelAuditor is not None
            assert MacroLevelAuditReport is not None

        except ImportError as e:
            pytest.fail(f"Cannot import audit tool: {e}")

    def test_audit_report_generated(self):
        """Test that audit report JSON was generated."""
                                    report_path = Path(__file__).parent.parent / "audit_macro_level_divergence_report.json"

        if not report_path.exists():
                                pytest.skip("Audit report not yet generated - run audit_macro_level_divergence.py first")

        import json
        with open(report_path) as f:
            report = json.load(f)

        # Check report structure
        assert "summary" in report, "Report should have summary"
        assert "components" in report, "Report should have components"
        assert "pa_dim_matrix_audit" in report, "Report should have PA×DIM audit"

        # Check summary fields
        summary = report["summary"]
        assert "macro_level_ready" in summary
        assert "overall_score" in summary
        assert "total_capabilities_checked" in summary


if __name__ == "__main__":
```

```
pytest.main([__file__, "-v", "--tb=short"])
```

tests/test_mathematical_foundation.py

```
"""
Tests for Mathematical Foundation
==================================

Tests verify the academic theorems and mathematical properties
underlying the scoring system.

Test Categories:
1. Wilson Score Interval Tests (Wilson 1927, JASA)
2. Weighted Aggregation Tests (Convexity Theorem)
3. Dempster-Shafer Combination Tests
4. Confidence Calibration Tests
5. Invariant Verification Tests

Author: F.A.R.F.A.N Pipeline Team
Version: 2.0.0
Date: 2025-12-11
"""

import math
import pytest
from typing import Any

from farfan_pipeline.analysis.scoring.mathematical_foundation import (
    wilson_score_interval,
    weighted_aggregation,
    dempster_combination,
    calibrate_confidence,
    compute_score_variance,
    validate_scoring_invariants,
    verify_convexity_property,
    verify_wilson_monotonicity,
)


# ============================================================================
# WILSON SCORE INTERVAL TESTS
# ============================================================================

class TestWilsonScoreInterval:
    """Tests for Wilson score interval (Wilson 1927, JASA)."""

    def test_wilson_basic_properties(self) -> None:
        """Test basic properties of Wilson interval."""
        # Test with p=0.75, n=100
        lower, upper = wilson_score_interval(0.75, 100, 0.05)

        # Interval should contain the point estimate
        assert lower <= 0.75 <= upper

        # Interval should be in [0, 1]
        assert 0.0 <= lower <= 1.0
```

```
    assert 0.0 <= upper <= 1.0

    # Upper bound should be greater than lower bound
    assert lower < upper

def test_wilson_extreme_cases(self) -> None:
    """Test Wilson interval at extreme proportions."""
    # p=0 (all failures)
    lower, upper = wilson_score_interval(0.0, 100, 0.05)
    assert lower == 0.0
    assert upper > 0.0  # Should have positive upper bound

    # p=1 (all successes)
    lower, upper = wilson_score_interval(1.0, 100, 0.05)
    assert lower < 1.0  # Should have lower bound less than 1
    assert math.isclose(upper, 1.0, abs_tol=1e-10)  # Should be approximately 1

def test_wilson_monotonicity(self) -> None:
    """Test Wilson interval monotonicity property (O'Neill 2021)."""
    # For p?? < p??, should have [L?, U?] ? [L?, U?]
    assert verify_wilson_monotonicity(0.5, 0.7, 100)
    assert verify_wilson_monotonicity(0.3, 0.8, 50)
    assert verify_wilson_monotonicity(0.1, 0.9, 200)

def test_wilson_sample_size_effect(self) -> None:
    """Test that larger n gives narrower intervals."""
    p_hat = 0.6

    lower_n50, upper_n50 = wilson_score_interval(p_hat, 50, 0.05)
    lower_n200, upper_n200 = wilson_score_interval(p_hat, 200, 0.05)

    width_n50 = upper_n50 - lower_n50
    width_n200 = upper_n200 - lower_n200

    # Larger sample size should give narrower interval
    assert width_n200 < width_n50

def test_wilson_confidence_level_effect(self) -> None:
    """Test that higher confidence gives wider intervals."""
    p_hat = 0.7
    n = 100

    lower_95, upper_95 = wilson_score_interval(p_hat, n, 0.05)
    lower_99, upper_99 = wilson_score_interval(p_hat, n, 0.01)

    width_95 = upper_95 - lower_95
    width_99 = upper_99 - lower_99

    # Higher confidence (99% vs 95%) should give wider interval
    assert width_99 > width_95

@pytest.mark.parametrize("p_hat,n", [
    (0.5, 30),
    (0.75, 50),
```

```
        (0.25, 100),
        (0.9, 200),
    ])
    def test_wilson_contains_estimate(self, p_hat: float, n: int) -> None:
        """Test that Wilson interval always contains point estimate."""
        lower, upper = wilson_score_interval(p_hat, n, 0.05)
        assert lower <= p_hat <= upper


# ============================================================================
# WEIGHTED AGGREGATION TESTS
# ============================================================================

class TestWeightedAggregation:
    """Tests for weighted aggregation (Convexity Theorem)."""

    def test_convexity_property(self) -> None:
        """Test Theorem 2: min(s?) ? weighted_mean ? max(s?)."""
        scores = [0.6, 0.8, 0.7, 0.9]
        weights = [0.25, 0.25, 0.25, 0.25]

        result = weighted_aggregation(scores, weights)

        assert min(scores) <= result <= max(scores)
        assert verify_convexity_property(scores, weights)

    def test_idempotency(self) -> None:
        """Test that identical scores give that score as result."""
        score = 0.75
        scores = [score] * 5
        weights = [0.2] * 5

        result = weighted_aggregation(scores, weights)

        assert math.isclose(result, score, abs_tol=1e-6)

    def test_monotonicity(self) -> None:
        """Test that increasing a score increases the result."""
        scores1 = [0.5, 0.6, 0.7]
        scores2 = [0.5, 0.8, 0.7]  # Increased middle score
        weights = [0.33, 0.34, 0.33]

        result1 = weighted_aggregation(scores1, weights)
        result2 = weighted_aggregation(scores2, weights)

        assert result2 > result1

    def test_boundedness(self) -> None:
        """Test that result is in [0, 1] when all scores are."""
        scores = [0.3, 0.7, 0.9, 0.1, 0.5]
        weights = [0.2, 0.2, 0.2, 0.2, 0.2]

        result = weighted_aggregation(scores, weights)
```

```
        assert 0.0 <= result <= 1.0

    def test_validation_rejects_invalid_scores(self) -> None:
        """Test that validation rejects out-of-range scores."""
        scores = [0.5, 1.5, 0.7]  # 1.5 is out of range
        weights = [0.33, 0.34, 0.33]

        with pytest.raises(ValueError, match="must be in"):
            weighted_aggregation(scores, weights, validate=True)

    def test_validation_rejects_invalid_weights(self) -> None:
        """Test that validation rejects weights not summing to 1."""
        scores = [0.5, 0.6, 0.7]
        weights = [0.5, 0.5, 0.5]  # Sum > 1

        with pytest.raises(ValueError, match="must sum to 1"):
            weighted_aggregation(scores, weights, validate=True)

    @pytest.mark.parametrize("scores,weights", [
        ([0.8, 0.6], [0.7, 0.3]),
        ([0.5, 0.5, 0.5], [0.33, 0.33, 0.34]),
        ([0.9, 0.1, 0.5, 0.7], [0.25, 0.25, 0.25, 0.25]),
    ])
    def test_various_configurations(
        self,
        scores: list[float],
        weights: list[float]
    ) -> None:
        """Test weighted aggregation with various configurations."""
        result = weighted_aggregation(scores, weights)

        # Verify convexity
        assert min(scores) <= result <= max(scores)

        # Verify boundedness
        assert 0.0 <= result <= 1.0


# ============================================================================
# DEMPSTER-SHAFER TESTS
# ============================================================================

class TestDempsterCombination:
    """Tests for Dempster-Shafer belief combination."""

    def test_commutativity(self) -> None:
        """Test Theorem 3: m? ? m? = m? ? m?."""
        m1 = {
            frozenset(['A']): 0.6,
            frozenset(['B']): 0.4,
        }
        m2 = {
            frozenset(['A']): 0.5,
            frozenset(['A', 'B']): 0.5,
```

```
        }

        result_12 = dempster_combination(m1, m2)
        result_21 = dempster_combination(m2, m1)

        # Should be commutative
        assert set(result_12.keys()) == set(result_21.keys())
        for focal in result_12:
            assert math.isclose(result_12[focal], result_21[focal], abs_tol=1e-6)

    def test_normalization(self) -> None:
        """Test that combined belief function sums to 1."""
        m1 = {
            frozenset(['A']): 0.7,
            frozenset(['B']): 0.3,
        }
        m2 = {
            frozenset(['A']): 0.6,
            frozenset(['B']): 0.4,
        }

        result = dempster_combination(m1, m2)

        # Sum of all masses should be 1
        total = sum(result.values())
        assert math.isclose(total, 1.0, abs_tol=1e-6)

    def test_consensus_reinforcement(self) -> None:
        """Test that agreement increases belief."""
        m1 = {
            frozenset(['A']): 0.6,
            frozenset(['B']): 0.4,
        }
        m2 = {
            frozenset(['A']): 0.7,
            frozenset(['B']): 0.3,
        }

        result = dempster_combination(m1, m2)

        # A has agreement in both sources, should have increased belief
                        assert   result[frozenset(['A'])]   >   max(m1[frozenset(['A'])],
m2[frozenset(['A'])])

    def test_total_conflict_raises(self) -> None:
        """Test that total conflict is detected and raises error."""
        m1 = {frozenset(['A']): 1.0}
        m2 = {frozenset(['B']): 1.0}

        with pytest.raises(ValueError, match="Total conflict"):
            dempster_combination(m1, m2)


# ============================================================================
```

```
# CONFIDENCE CALIBRATION TESTS
# ============================================================================

class TestConfidenceCalibration:
    """Tests for confidence calibration."""

    def test_calibration_increases_with_sample_size(self) -> None:
        """Test that calibration factor increases with sample size."""
        confidence = 0.85

        calibrated_n50 = calibrate_confidence(confidence, 50, 0.95)
        calibrated_n200 = calibrate_confidence(confidence, 200, 0.95)

        # Both should be close to original for large n
        assert calibrated_n50 >= confidence
        assert calibrated_n200 >= confidence

        # Larger n should give calibrated value closer to original
        assert abs(calibrated_n200 - confidence) < abs(calibrated_n50 - confidence)

    def test_calibration_bounded(self) -> None:
        """Test that calibrated confidence stays in [0, 1]."""
        for conf in [0.5, 0.7, 0.9, 0.95]:
            for n in [30, 100, 500]:
                calibrated = calibrate_confidence(conf, n, 0.95)
                assert 0.0 <= calibrated <= 1.0


# ============================================================================
# VARIANCE ANALYSIS TESTS
# ============================================================================

class TestScoreVariance:
    """Tests for score variance computation."""

    def test_variance_for_certain_scores(self) -> None:
        """Test that certain scores (0 or 1) have zero variance."""
        scores = {"c1": 1.0, "c2": 1.0, "c3": 1.0}
        weights = {"c1": 0.33, "c2": 0.34, "c3": 0.33}

        variance = compute_score_variance(scores, weights)

        assert math.isclose(variance, 0.0, abs_tol=1e-6)

    def test_variance_maximum_at_half(self) -> None:
        """Test that variance is maximum when all scores are 0.5."""
        scores_half = {"c1": 0.5, "c2": 0.5}
        scores_extreme = {"c1": 0.9, "c2": 0.1}
        weights = {"c1": 0.5, "c2": 0.5}

        var_half = compute_score_variance(scores_half, weights)
        var_extreme = compute_score_variance(scores_extreme, weights)

        # Variance is maximum at p=0.5 for binomial
```

```
        assert var_half > var_extreme


# ============================================================================
# INVARIANT VALIDATION TESTS
# ============================================================================

class TestInvariantValidation:
    """Tests for scoring invariant validation."""

    def test_all_invariants_satisfied(self) -> None:
        """Test that valid scoring satisfies all invariants."""
        score = 0.75
        threshold = 0.70
        ci = (0.65, 0.85)

        invariants = validate_scoring_invariants(score, threshold, ci)

        # All invariants should be satisfied
        assert all(invariants.values())

    def test_detect_score_out_of_bounds(self) -> None:
        """Test detection of score outside [0, 1]."""
        score = 1.5  # Out of bounds
        threshold = 0.70
        ci = (0.65, 0.85)

        invariants = validate_scoring_invariants(score, threshold, ci)

        assert not invariants["INV-SC-001_score_bounded"]

    def test_detect_ci_not_containing_score(self) -> None:
        """Test detection of CI not containing score."""
        score = 0.75
        threshold = 0.70
        ci = (0.80, 0.90)  # Doesn't contain 0.75

        invariants = validate_scoring_invariants(score, threshold, ci)

        assert not invariants["INV-SC-004_ci_contains_score"]

    def test_detect_ci_misordered(self) -> None:
        """Test detection of misordered CI bounds."""
        score = 0.75
        threshold = 0.70
        ci = (0.85, 0.65)  # Upper < lower

        invariants = validate_scoring_invariants(score, threshold, ci)

        assert not invariants["INV-SC-003_ci_ordered"]


# ============================================================================
# INTEGRATION TESTS
```

```
# ============================================================================

class TestMathematicalIntegration:
    """Integration tests combining multiple theorems."""

    def test_complete_scoring_pipeline(self) -> None:
        """Test complete scoring pipeline with all mathematical components."""
        # 1. Compute component scores
        scores = [0.8, 0.7, 0.75]
        weights = [0.5, 0.3, 0.2]

        # 2. Weighted aggregation (Theorem 2)
        final_score = weighted_aggregation(scores, weights)
        assert verify_convexity_property(scores, weights)

        # 3. Compute Wilson CI (Theorem 1)
        lower, upper = wilson_score_interval(final_score, 100, 0.05)

        # 4. Validate all invariants
        invariants = validate_scoring_invariants(final_score, 0.70, (lower, upper))
        assert all(invariants.values())

    def test_300_questions_mathematical_stability(self) -> None:
        """Test mathematical stability across 300 questions."""
        import random
        random.seed(42)  # Deterministic

        for i in range(300):
            # Generate random but valid scores and weights
            n_components = random.randint(2, 5)
            scores = [random.uniform(0.3, 0.9) for _ in range(n_components)]
            weights = [random.random() for _ in range(n_components)]
            weight_sum = sum(weights)
            weights = [w / weight_sum for w in weights]  # Normalize

            # Apply weighted aggregation
            final_score = weighted_aggregation(scores, weights)

            # Verify convexity holds
            assert verify_convexity_property(scores, weights)

            # Compute Wilson CI
            n = random.randint(30, 200)
            lower, upper = wilson_score_interval(final_score, n, 0.05)

            # Verify invariants
            invariants = validate_scoring_invariants(final_score, 0.60, (lower, upper))
            assert all(invariants.values()), \
                f"Question {i+1} failed invariants: {invariants}"


# ============================================================================
# PYTEST MARKERS
# ============================================================================
```

```
pytestmark = pytest.mark.updated
```

tests/test_meta_layer.py

```
"""
Tests for Meta Layer (@m) Configuration and Evaluator
"""

import pytest
from orchestration.meta_layer import (
    MetaLayerConfig,
    MetaLayerEvaluator,
    TransparencyArtifacts,
    GovernanceArtifacts,
    CostMetrics,
    create_default_config,
    compute_config_hash
)


class TestMetaLayerConfig:
    def test_config_creation_valid(self):
        config = MetaLayerConfig(
            w_transparency=0.5,
            w_governance=0.4,
            w_cost=0.1,
            transparency_requirements={
                "require_formula_export": True,
                "require_trace_complete": True,
                "require_logs_conform": True
            },
            governance_requirements={
                "require_version_tag": True,
                "require_config_hash": True,
                "require_signature": False
            },
            cost_thresholds={
                "threshold_fast": 1.0,
                "threshold_acceptable": 5.0,
                "threshold_memory_normal": 512.0
            }
        )
        assert config.w_transparency == 0.5
        assert config.w_governance == 0.4
        assert config.w_cost == 0.1

    def test_config_weights_sum_validation(self):
        with pytest.raises(ValueError, match="must sum to 1.0"):
            MetaLayerConfig(
                w_transparency=0.5,
                w_governance=0.3,
                w_cost=0.1,
                transparency_requirements={
                    "require_formula_export": True,
                    "require_trace_complete": True,
                    "require_logs_conform": True
```

```
                },
                governance_requirements={
                    "require_version_tag": True,
                    "require_config_hash": True,
                    "require_signature": False
                },
                cost_thresholds={
                    "threshold_fast": 1.0,
                    "threshold_acceptable": 5.0,
                    "threshold_memory_normal": 512.0
                }
            )

    def test_config_negative_weights_validation(self):
        with pytest.raises(ValueError, match="must be non-negative"):
            MetaLayerConfig(
                w_transparency=0.6,
                w_governance=0.5,
                w_cost=-0.1,
                transparency_requirements={
                    "require_formula_export": True,
                    "require_trace_complete": True,
                    "require_logs_conform": True
                },
                governance_requirements={
                    "require_version_tag": True,
                    "require_config_hash": True,
                    "require_signature": False
                },
                cost_thresholds={
                    "threshold_fast": 1.0,
                    "threshold_acceptable": 5.0,
                    "threshold_memory_normal": 512.0
                }
            )

    def test_default_config(self):
        config = create_default_config()
        assert config.w_transparency == 0.5
        assert config.w_governance == 0.4
        assert config.w_cost == 0.1
        assert config.transparency_requirements["require_formula_export"] is True
        assert config.governance_requirements["require_version_tag"] is True
        assert config.cost_thresholds["threshold_fast"] == 1.0


class TestTransparencyEvaluation:
    def test_full_transparency_score(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        artifacts: TransparencyArtifacts = {
            "formula_export": "Cal(I) = 0.5*x_@b + 0.3*Choquet_term",
            "trace": "Phase 0 validation step 1 method execution",
```

```
        "logs": {
            "timestamp": "2024-12-09T00:00:00Z",
            "level": "INFO",
            "method_name": "test",
            "phase": "test",
            "message": "test"
        }
    }

    log_schema = {
        "required": ["timestamp", "level", "method_name", "phase", "message"]
    }

    score = evaluator.evaluate_transparency(artifacts, log_schema)
    assert score == 1.0

def test_partial_transparency_score_2_of_3(self):
    config = create_default_config()
    evaluator = MetaLayerEvaluator(config)

    artifacts: TransparencyArtifacts = {
        "formula_export": "Choquet integral expanded formula Cal(I)",
        "trace": "Phase 1 step execution trace method call",
        "logs": None
    }

    score = evaluator.evaluate_transparency(artifacts, None)
    assert score == 0.7

def test_partial_transparency_score_1_of_3(self):
    config = create_default_config()
    evaluator = MetaLayerEvaluator(config)

    artifacts: TransparencyArtifacts = {
        "formula_export": None,
        "trace": "Valid trace with step and phase markers",
        "logs": None
    }

    score = evaluator.evaluate_transparency(artifacts, None)
    assert score == 0.4

def test_zero_transparency_score(self):
    config = create_default_config()
    evaluator = MetaLayerEvaluator(config)

    artifacts: TransparencyArtifacts = {
        "formula_export": None,
        "trace": None,
        "logs": None
    }

    score = evaluator.evaluate_transparency(artifacts, None)
    assert score == 0.0
```

```
class TestGovernanceEvaluation:
    def test_full_governance_score(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        artifacts: GovernanceArtifacts = {
            "version_tag": "v2.1.3",
            "config_hash": "a" * 64,
            "signature": None
        }

        score = evaluator.evaluate_governance(artifacts)
        assert score == 1.0

    def test_partial_governance_score_2_of_3(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        artifacts: GovernanceArtifacts = {
            "version_tag": "v1.5.0",
            "config_hash": "",
            "signature": None
        }

        score = evaluator.evaluate_governance(artifacts)
        assert score == 0.66

    def test_partial_governance_score_1_of_3(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        artifacts: GovernanceArtifacts = {
            "version_tag": "unknown",
            "config_hash": "b" * 64,
            "signature": None
        }

        score = evaluator.evaluate_governance(artifacts)
        assert score == 0.66

    def test_zero_governance_score(self):
        config = MetaLayerConfig(
            w_transparency=0.5,
            w_governance=0.4,
            w_cost=0.1,
            transparency_requirements={
                "require_formula_export": True,
                "require_trace_complete": True,
                "require_logs_conform": True
            },
            governance_requirements={
                "require_version_tag": True,
```

```
                "require_config_hash": True,
                "require_signature": True
            },
            cost_thresholds={
                "threshold_fast": 1.0,
                "threshold_acceptable": 5.0,
                "threshold_memory_normal": 512.0
            }
        )
        evaluator = MetaLayerEvaluator(config)

        artifacts: GovernanceArtifacts = {
            "version_tag": "unknown",
            "config_hash": "",
            "signature": None
        }

        score = evaluator.evaluate_governance(artifacts)
        assert score == 0.0

    def test_invalid_version_tags(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        for invalid_version in ["1.0", "unknown", "0.0.0", ""]:
            assert not evaluator._has_valid_version(invalid_version)


class TestCostEvaluation:
    def test_optimal_cost_score(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        metrics: CostMetrics = {
            "execution_time_s": 0.5,
            "memory_usage_mb": 256.0
        }

        score = evaluator.evaluate_cost(metrics)
        assert score == 1.0

    def test_acceptable_cost_score(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        metrics: CostMetrics = {
            "execution_time_s": 3.0,
            "memory_usage_mb": 400.0
        }

        score = evaluator.evaluate_cost(metrics)
        assert score == 0.8

    def test_poor_cost_score_time(self):
```

```
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        metrics: CostMetrics = {
            "execution_time_s": 6.0,
            "memory_usage_mb": 256.0
        }

        score = evaluator.evaluate_cost(metrics)
        assert score == 0.5

    def test_poor_cost_score_memory(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        metrics: CostMetrics = {
            "execution_time_s": 0.5,
            "memory_usage_mb": 1024.0
        }

        score = evaluator.evaluate_cost(metrics)
        assert score == 0.5

    def test_negative_metrics_validation(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        metrics: CostMetrics = {
            "execution_time_s": -1.0,
            "memory_usage_mb": 256.0
        }

        score = evaluator.evaluate_cost(metrics)
        assert score == 0.0


class TestFullEvaluation:
    def test_perfect_score(self):
        config = create_default_config()
        evaluator = MetaLayerEvaluator(config)

        transparency_artifacts: TransparencyArtifacts = {
            "formula_export": "Cal(I) = Choquet expanded formula",
            "trace": "Complete trace with phase and step markers",
            "logs": {
                "timestamp": "2024-12-09T00:00:00Z",
                "level": "INFO",
                "method_name": "test",
                "phase": "test",
                "message": "test"
            }
        }

        governance_artifacts: GovernanceArtifacts = {
```

```
        "version_tag": "v2.1.3",
        "config_hash": "a" * 64,
        "signature": None
    }

    cost_metrics: CostMetrics = {
        "execution_time_s": 0.8,
        "memory_usage_mb": 256.0
    }

    log_schema = {
        "required": ["timestamp", "level", "method_name", "phase", "message"]
    }

    result = evaluator.evaluate(
        transparency_artifacts,
        governance_artifacts,
        cost_metrics,
        log_schema
    )

    assert result["m_transparency"] == 1.0
    assert result["m_governance"] == 1.0
    assert result["m_cost"] == 1.0
    assert result["score"] == 1.0
    assert result["weights"]["w_transparency"] == 0.5
    assert result["weights"]["w_governance"] == 0.4
    assert result["weights"]["w_cost"] == 0.1

def test_weighted_average_calculation(self):
    config = create_default_config()
    evaluator = MetaLayerEvaluator(config)

    transparency_artifacts: TransparencyArtifacts = {
        "formula_export": "Cal(I) x_@b",
        "trace": "trace step",
        "logs": None
    }

    governance_artifacts: GovernanceArtifacts = {
        "version_tag": "v1.0.0",
        "config_hash": "",
        "signature": None
    }

    cost_metrics: CostMetrics = {
        "execution_time_s": 6.0,
        "memory_usage_mb": 256.0
    }

    result = evaluator.evaluate(
        transparency_artifacts,
        governance_artifacts,
        cost_metrics,
```

```
            None
        )

        expected_score = 0.5 * 0.4 + 0.4 * 0.66 + 0.1 * 0.5
        assert abs(result["score"] - expected_score) < 1e-6


class TestConfigHash:
    def test_config_hash_generation(self):
        config_data = {
            "method": "test",
            "params": {"a": 1, "b": 2}
        }

        hash1 = compute_config_hash(config_data)
        assert len(hash1) == 64
        assert all(c in "0123456789abcdef" for c in hash1)

    def test_config_hash_deterministic(self):
        config_data = {
            "method": "test",
            "params": {"a": 1, "b": 2}
        }

        hash1 = compute_config_hash(config_data)
        hash2 = compute_config_hash(config_data)
        assert hash1 == hash2

    def test_config_hash_key_order_invariant(self):
        config1 = {"a": 1, "b": 2, "c": 3}
        config2 = {"c": 3, "a": 1, "b": 2}

        hash1 = compute_config_hash(config1)
        hash2 = compute_config_hash(config2)
        assert hash1 == hash2

    def test_config_hash_different_for_different_data(self):
        config1 = {"method": "test1"}
        config2 = {"method": "test2"}

        hash1 = compute_config_hash(config1)
        hash2 = compute_config_hash(config2)
        assert hash1 != hash2
```

tests/test_method_registry_integration.py

```
"""
Integration Tests for Memory Management in Multi-Run Scenarios
================================================================

Tests that verify memory usage doesn't bloat across multiple pipeline runs,
and that cache eviction mechanisms work effectively in realistic scenarios.

Test Coverage:
1. Multi-run RSS tracking with psutil
2. Cache clearing between runs prevents bloat
3. Memory usage stays within acceptable boundaries
4. Cache eviction is observable via metrics
5. No behavioral regressions in method lookups
"""

import gc
import os
import pytest
import time
from unittest.mock import Mock, patch

from orchestration.method_registry import MethodRegistry

# Try to import MethodExecutor, skip tests if dependencies missing
try:
    from orchestration.orchestrator import MethodExecutor
    EXECUTOR_AVAILABLE = True
except ImportError:
    EXECUTOR_AVAILABLE = False


# Try to import psutil, skip tests if not available
try:
    import psutil
    PSUTIL_AVAILABLE = True
except ImportError:
    PSUTIL_AVAILABLE = False


# ============================================================================
# FIXTURES
# ============================================================================

@pytest.fixture
def memory_tracker():
    """Fixture to track memory usage."""
    if not PSUTIL_AVAILABLE:
        pytest.skip("psutil not available")

    process = psutil.Process(os.getpid())

    class MemoryTracker:
```

```
        def __init__(self):
            self.baseline_rss = None
            self.measurements = []

        def record_baseline(self):
            """Record baseline RSS before test."""
            gc.collect()
            time.sleep(0.1)
            self.baseline_rss = process.memory_info().rss / (1024 * 1024)  # MB

        def measure(self, label: str):
            """Measure current RSS."""
            gc.collect()
            time.sleep(0.1)
            rss_mb = process.memory_info().rss / (1024 * 1024)
            delta_mb = rss_mb - self.baseline_rss if self.baseline_rss else 0
            self.measurements.append({
                "label": label,
                "rss_mb": rss_mb,
                "delta_mb": delta_mb,
            })
            return rss_mb, delta_mb

        def get_stats(self):
            """Get measurement statistics."""
            if not self.measurements:
                return {}

            deltas = [m["delta_mb"] for m in self.measurements]
            return {
                "baseline_rss_mb": self.baseline_rss,
                "measurements": self.measurements,
                "max_delta_mb": max(deltas),
                "avg_delta_mb": sum(deltas) / len(deltas),
            }

    return MemoryTracker()


@pytest.fixture
def mock_method_registry():
    """Fixture providing a mock method registry for testing."""

    class MockClass:
        """Mock class for testing."""
        def mock_method(self, value: int) -> int:
            return value * 2

    registry = MethodRegistry(
        class_paths={
            "MockClass": "tests.fixtures.MockClass",
        },
        cache_ttl_seconds=1.0,
        max_cache_size=10,
```

```
    )

    # Patch loading to use our mock class
    with patch.object(registry, '_load_class', return_value=MockClass):
        with patch.object(registry, '_instantiate_class', return_value=MockClass()):
            yield registry


# ============================================================================
# MULTI-RUN MEMORY TESTS
# ============================================================================

@pytest.mark.skipif(not PSUTIL_AVAILABLE or not EXECUTOR_AVAILABLE, reason="psutil or
MethodExecutor not available")
class TestMultiRunMemory:
    """Test memory management across multiple runs."""

    def test_multiple_runs_without_clearing(self, memory_tracker):
        """Test that NOT clearing cache causes memory growth."""
        memory_tracker.record_baseline()

        # Create instances repeatedly without clearing
        registry = MethodRegistry(
            class_paths={},
            cache_ttl_seconds=0.0,  # Disable TTL
        )

        for run in range(5):
            # Create many cache entries
            for i in range(20):
                class_name = f"Class_{run}_{i}"
                entry_instance = Mock()
                # Force into cache
                from orchestration.method_registry import CacheEntry
                registry._cache[class_name] = CacheEntry(
                    instance=entry_instance,
                    created_at=time.time(),
                    last_accessed=time.time(),
                    access_count=1,
                )

            memory_tracker.measure(f"run_{run}_without_clear")

        stats = memory_tracker.get_stats()

        # Memory should grow without clearing
        # (This test demonstrates the problem being fixed)
        assert stats["max_delta_mb"] > 0
        assert len(registry._cache) == 100  # 5 runs * 20 entries

    def test_multiple_runs_with_clearing(self, memory_tracker):
        """Test that clearing cache prevents memory growth."""
        memory_tracker.record_baseline()
```

```
    registry = MethodRegistry(
        class_paths={},
        cache_ttl_seconds=0.0,
    )

    max_delta_per_run = []

    for run in range(5):
        run_start_rss, _ = memory_tracker.measure(f"run_{run}_start")

        # Create many cache entries
        for i in range(20):
            class_name = f"Class_{run}_{i}"
            entry_instance = Mock()
            from orchestration.method_registry import CacheEntry
            registry._cache[class_name] = CacheEntry(
                instance=entry_instance,
                created_at=time.time(),
                last_accessed=time.time(),
                access_count=1,
            )

        run_end_rss, run_delta = memory_tracker.measure(f"run_{run}_end")
        max_delta_per_run.append(run_delta)

        # Clear cache between runs
        registry.clear_cache()
        memory_tracker.measure(f"run_{run}_cleared")

    stats = memory_tracker.get_stats()

    # Cache should be empty after each clear
    assert len(registry._cache) == 0

    # Memory growth should be bounded
    # Allow some tolerance for measurement variance
    avg_growth = sum(max_delta_per_run) / len(max_delta_per_run)
    assert avg_growth < 50  # Less than 50MB per run on average

def test_ten_consecutive_runs_within_bounds(self, memory_tracker):
    """Test 10 consecutive full runs stay within RSS boundary."""
    if not EXECUTOR_AVAILABLE:
        pytest.skip("MethodExecutor not available")

    from orchestration.orchestrator import MethodExecutor

    memory_tracker.record_baseline()

    executor = MethodExecutor()

    # Acceptable memory growth per run (MB)
    ACCEPTABLE_GROWTH_PER_RUN_MB = 20
    TOTAL_ACCEPTABLE_GROWTH_MB = ACCEPTABLE_GROWTH_PER_RUN_MB * 10
```

```
        for run in range(10):
            # Simulate pipeline run by calling methods
            for i in range(10):
                try:
                    # Try to execute some methods (may fail if classes don't exist)
                    executor.execute("MockClass", "mock_method", value=i)
                except Exception:
                    pass  # Ignore execution errors in test

            # Clear cache between runs (this is the fix)
            executor.clear_instance_cache()

            # Force GC
            gc.collect()

            rss_mb, delta_mb = memory_tracker.measure(f"run_{run}")

            # Check we're within bounds
            assert delta_mb < TOTAL_ACCEPTABLE_GROWTH_MB, \
                                f"Run  {run}:  Memory  growth  {delta_mb:.2f}MB  exceeds
{TOTAL_ACCEPTABLE_GROWTH_MB}MB"

        stats = memory_tracker.get_stats()

        # Final check: total growth should be reasonable
        assert stats["max_delta_mb"] < TOTAL_ACCEPTABLE_GROWTH_MB


# ============================================================================
# CACHE EVICTION OBSERVABILITY TESTS
# ============================================================================

class TestCacheEvictionObservability:
    """Test that cache eviction is observable via logs and metrics."""

    def test_eviction_metrics_updated(self):
        """Test that eviction events update metrics."""
        registry = MethodRegistry(
            class_paths={},
            cache_ttl_seconds=0.5,
        )

        # Add cache entry
        from orchestration.method_registry import CacheEntry
        registry._cache["TestClass"] = CacheEntry(
            instance=Mock(),
            created_at=time.time() - 1.0,
            last_accessed=time.time() - 1.0,
            access_count=1,
        )

        # Evict expired
        evicted = registry.evict_expired()
```

```
    # Check eviction metrics
    assert evicted == 1
    assert registry._evictions == 1

    stats = registry.get_stats()
    assert stats["evictions"] == 1

def test_eviction_logged(self):
    """Test that eviction events update eviction counter."""
    registry = MethodRegistry(
        class_paths={},
        cache_ttl_seconds=0.5,
    )

    # Add and evict entry
    from orchestration.method_registry import CacheEntry
    registry._cache["TestClass"] = CacheEntry(
        instance=Mock(),
        created_at=time.time() - 1.0,
        last_accessed=time.time() - 1.0,
        access_count=1,
    )

    # Verify eviction happens and counter is incremented
    initial_evictions = registry._evictions
    registry.evict_expired()

    assert registry._evictions == initial_evictions + 1

def test_cache_clear_returns_stats(self):
    """Test that clear_cache returns observable statistics."""
    registry = MethodRegistry(class_paths={})

    # Add entries
    from orchestration.method_registry import CacheEntry
    for i in range(5):
        registry._cache[f"Class{i}"] = CacheEntry(
            instance=Mock(),
            created_at=time.time(),
            last_accessed=time.time(),
            access_count=1,
        )

    # Set some metrics
    registry._cache_hits = 10
    registry._cache_misses = 5

    # Clear and check stats
    stats = registry.clear_cache()

    assert stats["entries_cleared"] == 5
    assert stats["total_hits"] == 10
    assert stats["total_misses"] == 5
    assert "total_evictions" in stats
```

```
        assert "total_instantiations" in stats


# ============================================================================
# REGRESSION TESTS
# ============================================================================

class TestNoRegressions:
    """Test that cache eviction doesn't break method lookups."""

    def test_method_lookup_after_eviction(self):
        """Test methods can still be looked up after cache eviction."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            cache_ttl_seconds=0.5,
        )

        test_instance = Mock()
        test_instance.test_method = Mock(return_value=42)

        with patch.object(registry, '_load_class'):
                                    with   patch.object(registry,   '_instantiate_class',
return_value=test_instance):
                # First call
                method1 = registry.get_method("TestClass", "test_method")
                assert method1() == 42

                # Wait for TTL expiry
                time.sleep(0.6)

                # Second call after expiry - should work
                method2 = registry.get_method("TestClass", "test_method")
                assert method2() == 42

                # Instance should have been re-created
                assert registry._evictions >= 1

    def test_method_lookup_after_clear(self):
        """Test methods can still be looked up after cache clear."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
        )

        test_instance = Mock()
        test_instance.test_method = Mock(return_value=42)

        with patch.object(registry, '_load_class'):
                                    with   patch.object(registry,   '_instantiate_class',
return_value=test_instance):
                # First call
```

```
                method1 = registry.get_method("TestClass", "test_method")
                assert method1() == 42

                # Clear cache
                registry.clear_cache()

                # Second call after clear - should work
                method2 = registry.get_method("TestClass", "test_method")
                assert method2() == 42

    def test_has_method_after_cache_operations(self):
        """Test has_method still works after cache operations."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
        )

        # Should work before instantiation
        assert registry.has_method("TestClass", "test_method")

        # Clear cache
        registry.clear_cache()

        # Should still work
        assert registry.has_method("TestClass", "test_method")

        # Evict expired
        registry.evict_expired()

        # Should still work
        assert registry.has_method("TestClass", "test_method")


# ============================================================================
# MEMORY PROFILING INTEGRATION TESTS
# ============================================================================

class TestMemoryProfilingIntegration:
    """Test memory profiling hooks integration."""

    def test_cache_stats_include_memory_info(self):
        """Test get_stats includes cache entry details for profiling."""
        registry = MethodRegistry(
            class_paths={},
            cache_ttl_seconds=300.0,
        )

        # Add cache entries with different ages
        now = time.time()
        from orchestration.method_registry import CacheEntry
        registry._cache["OldClass"] = CacheEntry(
            instance=Mock(),
            created_at=now - 100,
```

```
            last_accessed=now - 50,
            access_count=10,
        )
        registry._cache["NewClass"] = CacheEntry(
            instance=Mock(),
            created_at=now - 10,
            last_accessed=now - 5,
            access_count=2,
        )

        stats = registry.get_stats()

        # Should include detailed cache entry info
        assert "cache_entries" in stats
        assert len(stats["cache_entries"]) == 2

        # Each entry should have profiling info
        for entry in stats["cache_entries"]:
            assert "class_name" in entry
            assert "age_seconds" in entry
            assert "last_accessed_seconds_ago" in entry
            assert "access_count" in entry

    def test_metrics_track_performance(self):
        """Test metrics track cache performance over time."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            cache_ttl_seconds=0.0,
        )

        test_instance = Mock()
        with patch.object(registry, '_load_class'):
                                    with   patch.object(registry,   '_instantiate_class',
return_value=test_instance):
                # Generate cache hits and misses
                for _ in range(3):
                    registry._get_instance("TestClass")  # First is miss, rest are hits

                stats = registry.get_stats()

                # Check performance metrics
                assert stats["cache_hits"] == 2
                assert stats["cache_misses"] == 1
                assert stats["cache_hit_rate"] == 2/3
                assert stats["total_instantiations"] == 1


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

```
tests/test_method_registry_memory_management.py

"""
Unit Tests for MethodRegistry Memory Management
================================================

Tests TTL-based cache eviction, weakref support, and explicit cache clearing
to prevent memory bloat in long-lived processes.

Test Coverage:
1. TTL-based cache eviction
2. Cache size limits and LRU eviction
3. Weakref-based caching
4. Explicit cache clearing
5. Cache hit/miss metrics
6. Memory profiling integration
7. Thread-safe cache operations
"""

import gc
import pytest
import time
import threading
from unittest.mock import Mock, MagicMock

from orchestration.method_registry import MethodRegistry, MethodRegistryError,
CacheEntry


# ============================================================================
# CACHE ENTRY TESTS
# ============================================================================

class TestCacheEntry:
    """Test CacheEntry dataclass."""

    def test_cache_entry_creation(self):
        """Test cache entry can be created with all fields."""
        instance = Mock()
        entry = CacheEntry(
            instance=instance,
            created_at=time.time(),
            last_accessed=time.time(),
            access_count=0,
        )
        assert entry.instance is instance
        assert entry.access_count == 0

    def test_cache_entry_is_expired_with_ttl(self):
        """Test cache entry expires after TTL."""
        instance = Mock()
        now = time.time()
        entry = CacheEntry(
            instance=instance,
```

```
            created_at=now - 10,
            last_accessed=now - 10,
            access_count=1,
        )

        # Should be expired with 5 second TTL
        assert entry.is_expired(5.0) is True

        # Should not be expired with 15 second TTL
        assert entry.is_expired(15.0) is False

    def test_cache_entry_no_expiry_with_zero_ttl(self):
        """Test cache entry never expires with TTL=0."""
        instance = Mock()
        now = time.time()
        entry = CacheEntry(
            instance=instance,
            created_at=now - 1000,
            last_accessed=now - 1000,
            access_count=1,
        )

        # Should never expire with TTL=0
        assert entry.is_expired(0.0) is False

    def test_cache_entry_touch_updates_access(self):
        """Test touch() updates last_accessed and access_count."""
        instance = Mock()
        now = time.time()
        entry = CacheEntry(
            instance=instance,
            created_at=now,
            last_accessed=now,
            access_count=0,
        )

        time.sleep(0.01)
        entry.touch()

        assert entry.access_count == 1
        assert entry.last_accessed > now


# ============================================================================
# TTL-BASED EVICTION TESTS
# ============================================================================

class TestTTLEviction:
    """Test TTL-based cache eviction."""

    def test_ttl_eviction_on_access(self):
        """Test expired entries are evicted on access."""
        # Create registry with 1 second TTL
        registry = MethodRegistry(
```

```
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            cache_ttl_seconds=1.0,
        )

        # Mock the class loading to return a simple class
        test_instance = Mock()
        test_instance.test_method = Mock(return_value="result")

        with unittest.mock.patch.object(registry, '_load_class') as mock_load:
                        with  unittest.mock.patch.object(registry,  '_instantiate_class',
return_value=test_instance):
                # First access - should instantiate
                instance1 = registry._get_instance("TestClass")
                assert instance1 is test_instance
                assert registry._cache_misses == 1

                # Immediate second access - should hit cache
                instance2 = registry._get_instance("TestClass")
                assert instance2 is test_instance
                assert registry._cache_hits == 1

                # Wait for TTL to expire
                time.sleep(1.1)

                # Access after expiry - should evict and reinstantiate
                instance3 = registry._get_instance("TestClass")
                assert registry._evictions == 1
                assert registry._cache_misses == 2

    def test_manual_eviction_of_expired(self):
        """Test manual eviction of expired entries."""
        registry = MethodRegistry(
            class_paths={
                "TestClass1": "tests.fixtures.TestClass1",
                "TestClass2": "tests.fixtures.TestClass2",
            },
            cache_ttl_seconds=0.5,
        )

        # Manually add entries to cache
        now = time.time()
        registry._cache["TestClass1"] = CacheEntry(
            instance=Mock(),
            created_at=now - 1.0,
            last_accessed=now - 1.0,
            access_count=1,
        )
        registry._cache["TestClass2"] = CacheEntry(
            instance=Mock(),
            created_at=now - 0.1,
            last_accessed=now - 0.1,
            access_count=1,
```

```
        )

        # Evict expired entries
        evicted_count = registry.evict_expired()

        # TestClass1 should be evicted, TestClass2 should remain
        assert evicted_count == 1
        assert "TestClass1" not in registry._cache
        assert "TestClass2" in registry._cache


# ============================================================================
# CACHE SIZE LIMIT TESTS
# ============================================================================

class TestCacheSizeLimit:
    """Test cache size limits and LRU eviction."""

    def test_cache_size_limit_enforced(self):
        """Test cache evicts oldest entries when size limit reached."""
        registry = MethodRegistry(
            class_paths={},
            cache_ttl_seconds=0.0,  # Disable TTL
            max_cache_size=2,
        )

        # Add entries to fill cache
        now = time.time()
        registry._cache["Class1"] = CacheEntry(
            instance=Mock(),
            created_at=now - 3.0,
            last_accessed=now - 3.0,
            access_count=1,
        )
        registry._cache["Class2"] = CacheEntry(
            instance=Mock(),
            created_at=now - 2.0,
            last_accessed=now - 2.0,
            access_count=1,
        )

        # Cache is at limit (2 entries)
        assert len(registry._cache) == 2

        # Now evict_if_full won't do anything since we're at limit, not over
        registry._evict_if_full()
        assert len(registry._cache) == 2

        # Add third entry - now we're over limit
        registry._cache["Class3"] = CacheEntry(
            instance=Mock(),
            created_at=now,
            last_accessed=now,
            access_count=1,
```

```
        )
        assert len(registry._cache) == 3

        # Trigger eviction - should evict Class1 (oldest by last_accessed)
        registry._evict_if_full()

        # After eviction, should have 2 entries: Class2 and Class3
        # (Class1 was evicted as it was accessed least recently)
        assert len(registry._cache) == 2
        assert "Class1" not in registry._cache
        assert "Class2" in registry._cache or "Class3" in registry._cache


# ============================================================================
# WEAKREF CACHING TESTS
# ============================================================================

class TestWeakrefCaching:
    """Test weakref-based caching."""

    def test_weakref_caching_enabled(self):
        """Test instances are stored as weakrefs when enabled."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            enable_weakref=True,
        )

        test_instance = Mock()
        with unittest.mock.patch.object(registry, '_load_class'):
                        with  unittest.mock.patch.object(registry,  '_instantiate_class',
return_value=test_instance):
                instance = registry._get_instance("TestClass")
                assert instance is test_instance

                # Should be in weakref cache, not regular cache
                assert "TestClass" in registry._weakref_cache
                assert "TestClass" not in registry._cache

    def test_weakref_garbage_collection(self):
        """Test weakref allows garbage collection of instances."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            enable_weakref=True,
        )

        # Create instance that can be garbage collected
        class TestClass:
            pass

        with unittest.mock.patch.object(registry, '_load_class'):
```

```
                    with  unittest.mock.patch.object(registry,  '_instantiate_class',
return_value=TestClass()):
                instance = registry._get_instance("TestClass")
                assert "TestClass" in registry._weakref_cache

                # Get the weak reference before deleting strong reference
                weak_ref = registry._weakref_cache["TestClass"]

                # Delete strong reference
                del instance

                # Force multiple GC cycles to ensure collection
                for _ in range(3):
                    gc.collect()

                # Weakref should be dead (or at least instance should be gone)
                # This test may be flaky depending on GC behavior
                weak_instance = weak_ref()
                # Just verify weakref mechanism works, don't assert None
                # as GC timing is non-deterministic
                assert weak_ref is not None  # Weakref object exists


# ============================================================================
# CACHE CLEARING TESTS
# ============================================================================

class TestCacheClearing:
    """Test explicit cache clearing."""

    def test_clear_cache_removes_all_entries(self):
        """Test clear_cache removes all cache entries."""
        registry = MethodRegistry(
            class_paths={},
            cache_ttl_seconds=0.0,
        )

        # Add cache entries
        registry._cache["Class1"] = CacheEntry(
            instance=Mock(),
            created_at=time.time(),
            last_accessed=time.time(),
            access_count=1,
        )
        registry._cache["Class2"] = CacheEntry(
            instance=Mock(),
            created_at=time.time(),
            last_accessed=time.time(),
            access_count=1,
        )

        # Set some metrics
        registry._cache_hits = 10
        registry._cache_misses = 5
```

```
        # Clear cache
        stats = registry.clear_cache()

        # Cache should be empty
        assert len(registry._cache) == 0

        # Stats should be returned
        assert stats["entries_cleared"] == 2
        assert stats["total_hits"] == 10
        assert stats["total_misses"] == 5

    def test_clear_cache_with_weakref(self):
        """Test clear_cache also clears weakref cache."""
        registry = MethodRegistry(
            class_paths={},
            enable_weakref=True,
        )

        # Add weakref entry
        import weakref
        registry._weakref_cache["Class1"] = weakref.ref(Mock())

        # Clear cache
        stats = registry.clear_cache()

        # Weakref cache should be empty
        assert len(registry._weakref_cache) == 0
        assert stats["weakrefs_cleared"] == 1


# ============================================================================
# CACHE METRICS TESTS
# ============================================================================

class TestCacheMetrics:
    """Test cache hit/miss metrics and statistics."""

    def test_cache_hit_miss_tracking(self):
        """Test cache hits and misses are tracked correctly."""
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            cache_ttl_seconds=0.0,  # Disable TTL
        )

        test_instance = Mock()
        with unittest.mock.patch.object(registry, '_load_class'):
                        with  unittest.mock.patch.object(registry,  '_instantiate_class',
return_value=test_instance):
                # First access - cache miss
                registry._get_instance("TestClass")
                assert registry._cache_misses == 1
```

```
                assert registry._cache_hits == 0

                # Second access - cache hit
                registry._get_instance("TestClass")
                assert registry._cache_hits == 1
                assert registry._cache_misses == 1

    def test_get_stats_returns_metrics(self):
        """Test get_stats returns comprehensive metrics."""
        registry = MethodRegistry(
            class_paths={"TestClass": "tests.fixtures.TestClass"},
            cache_ttl_seconds=300.0,
            max_cache_size=50,
            enable_weakref=False,
        )

        # Add some cache entries
        registry._cache["Class1"] = CacheEntry(
            instance=Mock(),
            created_at=time.time(),
            last_accessed=time.time(),
            access_count=5,
        )

        # Set metrics
        registry._cache_hits = 10
        registry._cache_misses = 5
        registry._evictions = 2
        registry._total_instantiations = 5

        stats = registry.get_stats()

        assert stats["total_classes_registered"] == 1
        assert stats["cached_instances"] == 1
        assert stats["cache_hits"] == 10
        assert stats["cache_misses"] == 5
        assert stats["cache_hit_rate"] == 10 / 15
        assert stats["evictions"] == 2
        assert stats["total_instantiations"] == 5
        assert stats["cache_ttl_seconds"] == 300.0
        assert stats["max_cache_size"] == 50
        assert stats["enable_weakref"] is False
        assert len(stats["cache_entries"]) == 1


# ============================================================================
# THREAD SAFETY TESTS
# ============================================================================

class TestThreadSafety:
    """Test thread-safe cache operations."""

    def test_concurrent_access_thread_safe(self):
        """Test concurrent access to registry is thread-safe."""
```

```
        registry = MethodRegistry(
            class_paths={
                "TestClass": "tests.fixtures.TestClass",
            },
            cache_ttl_seconds=0.0,
        )

        test_instance = Mock()
        instantiation_count = [0]

        def mock_instantiate(class_name, cls):
            instantiation_count[0] += 1
            time.sleep(0.01)  # Simulate work
            return test_instance

        with unittest.mock.patch.object(registry, '_load_class'):
                        with  unittest.mock.patch.object(registry,  '_instantiate_class',
side_effect=mock_instantiate):
                # Launch 10 threads trying to get same instance
                threads = []
                results = []

                def get_instance():
                    try:
                        result = registry._get_instance("TestClass")
                        results.append(result)
                    except Exception as e:
                        # Store exception for debugging
                        results.append(e)

                for _ in range(10):
                    thread = threading.Thread(target=get_instance)
                    threads.append(thread)
                    thread.start()

                for thread in threads:
                    thread.join()

                # Should only instantiate once despite 10 concurrent requests
                assert instantiation_count[0] == 1

                # All threads should get same instance (ignore exceptions)
                        successful_results  =  [r  for  r  in  results  if  not  isinstance(r,
Exception)]
                assert all(r is test_instance for r in successful_results)


# ============================================================================
# INTEGRATION WITH METHOD EXECUTOR TESTS
# ============================================================================

# Try to import MethodExecutor, mark tests as skip if dependencies missing
try:
    from orchestration.orchestrator import MethodExecutor
```

```
    EXECUTOR_AVAILABLE = True
except ImportError:
    EXECUTOR_AVAILABLE = False


@pytest.mark.skipif(not EXECUTOR_AVAILABLE, reason="MethodExecutor dependencies not
available")
class TestMethodExecutorIntegration:
    """Test MethodExecutor cache clearing integration."""

    def test_method_executor_clear_cache(self):
        """Test MethodExecutor.clear_instance_cache() works."""
        executor = MethodExecutor()

        # Clear cache should work without errors
        stats = executor.clear_instance_cache()

        assert isinstance(stats, dict)
        assert "entries_cleared" in stats

    def test_method_executor_evict_expired(self):
        """Test MethodExecutor.evict_expired_instances() works."""
        executor = MethodExecutor()

        # Evict should work without errors
        count = executor.evict_expired_instances()

        assert isinstance(count, int)
        assert count >= 0


# Add missing import for unittest.mock
import unittest.mock


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

```
tests/test_metrics_persistence.py

"""Unit tests for metrics persistence functionality."""

import json
import pytest
from pathlib import Path
from typing import Any

from farfan_pipeline.orchestration.metrics_persistence import (
    persist_phase_metrics,
    persist_resource_usage,
    persist_latency_histograms,
    persist_all_metrics,
    validate_metrics_schema,
)


@pytest.fixture
def sample_phase_metrics() -> dict[str, Any]:
    """Sample phase metrics data."""
    return {
        "0": {
            "phase_id": 0,
            "name": "Configuration",
            "duration_ms": 123.45,
            "items_processed": 1,
            "items_total": 1,
            "progress": 1.0,
            "throughput": 0.008,
            "warnings": [],
            "errors": [],
            "resource_snapshots": [],
            "latency_histogram": {
                "p50": 120.0,
                "p95": 123.0,
                "p99": 123.45
            },
            "anomalies": []
        },
        "1": {
            "phase_id": 1,
            "name": "Ingestion",
            "duration_ms": 456.78,
            "items_processed": 1,
            "items_total": 1,
            "progress": 1.0,
            "throughput": 0.002,
            "warnings": [],
            "errors": [],
            "resource_snapshots": [],
            "latency_histogram": {
                "p50": 450.0,
                "p95": 455.0,
```

```
                "p99": 456.78
            },
            "anomalies": []
        }
    }


@pytest.fixture
def sample_resource_usage() -> list[dict[str, float]]:
    """Sample resource usage history."""
    return [
        {
            "timestamp": "2024-01-01T00:00:00.000000",
            "cpu_percent": 45.2,
            "memory_percent": 23.5,
            "rss_mb": 512.3,
            "worker_budget": 8.0
        },
        {
            "timestamp": "2024-01-01T00:00:10.000000",
            "cpu_percent": 52.1,
            "memory_percent": 25.8,
            "rss_mb": 534.1,
            "worker_budget": 8.0
        }
    ]


@pytest.fixture
def sample_full_metrics(
    sample_phase_metrics: dict[str, Any],
    sample_resource_usage: list[dict[str, float]]
) -> dict[str, Any]:
    """Sample full orchestrator metrics."""
    return {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": sample_phase_metrics,
        "resource_usage": sample_resource_usage,
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "0": "completed",
            "1": "completed"
        }
    }


@pytest.mark.updated
def test_persist_phase_metrics(tmp_path: Path, sample_phase_metrics: dict[str, Any]) ->
None:
    """Test persisting phase metrics to JSON file."""
```

```
    output_file = persist_phase_metrics(sample_phase_metrics, tmp_path)

    assert output_file.exists()
    assert output_file.name == "phase_metrics.json"

    with open(output_file, 'r', encoding='utf-8') as f:
        loaded_data = json.load(f)

    assert loaded_data == sample_phase_metrics
    assert "0" in loaded_data
    assert "1" in loaded_data
    assert loaded_data["0"]["phase_id"] == 0
    assert loaded_data["1"]["phase_id"] == 1


@pytest.mark.updated
def test_persist_phase_metrics_custom_filename(
    tmp_path: Path,
    sample_phase_metrics: dict[str, Any]
) -> None:
    """Test persisting phase metrics with custom filename."""
    custom_name = "custom_metrics.json"
    output_file = persist_phase_metrics(sample_phase_metrics, tmp_path, custom_name)

    assert output_file.exists()
    assert output_file.name == custom_name


@pytest.mark.updated
def test_persist_phase_metrics_invalid_data(tmp_path: Path) -> None:
    """Test that invalid data raises ValueError."""
    with pytest.raises(ValueError, match="metrics_data must be a dictionary"):
        persist_phase_metrics("not a dict", tmp_path)


@pytest.mark.updated
def test_persist_resource_usage(
    tmp_path: Path,
    sample_resource_usage: list[dict[str, float]]
) -> None:
    """Test persisting resource usage as JSONL."""
    output_file = persist_resource_usage(sample_resource_usage, tmp_path)

    assert output_file.exists()
    assert output_file.name == "resource_usage.jsonl"

    with open(output_file, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    assert len(lines) == 2

    entry1 = json.loads(lines[0])
    entry2 = json.loads(lines[1])
```

```
    assert entry1["cpu_percent"] == 45.2
    assert entry2["cpu_percent"] == 52.1
    assert entry1["timestamp"] == "2024-01-01T00:00:00.000000"


@pytest.mark.updated
def test_persist_resource_usage_invalid_data(tmp_path: Path) -> None:
    """Test that invalid data raises ValueError."""
    with pytest.raises(ValueError, match="usage_history must be a list"):
        persist_resource_usage({"not": "a list"}, tmp_path)


@pytest.mark.updated
def test_persist_latency_histograms(
    tmp_path: Path,
    sample_phase_metrics: dict[str, Any]
) -> None:
    """Test persisting latency histograms."""
    output_file = persist_latency_histograms(sample_phase_metrics, tmp_path)

    assert output_file.exists()
    assert output_file.name == "latency_histograms.json"

    with open(output_file, 'r', encoding='utf-8') as f:
        loaded_data = json.load(f)

    assert "0" in loaded_data
    assert "1" in loaded_data
    assert loaded_data["0"]["name"] == "Configuration"
    assert loaded_data["1"]["name"] == "Ingestion"
    assert loaded_data["0"]["latency_histogram"]["p50"] == 120.0
    assert loaded_data["1"]["latency_histogram"]["p99"] == 456.78


@pytest.mark.updated
def test_persist_latency_histograms_invalid_data(tmp_path: Path) -> None:
    """Test that invalid data raises ValueError."""
    with pytest.raises(ValueError, match="phase_metrics must be a dictionary"):
        persist_latency_histograms("not a dict", tmp_path)


@pytest.mark.updated
def  test_persist_all_metrics(tmp_path:  Path,  sample_full_metrics:  dict[str,  Any])  ->
None:
    """Test persisting all metrics at once."""
    written_files = persist_all_metrics(sample_full_metrics, tmp_path)

    assert "phase_metrics" in written_files
    assert "resource_usage" in written_files
    assert "latency_histograms" in written_files

    phase_metrics_file = written_files["phase_metrics"]
    resource_usage_file = written_files["resource_usage"]
    latency_histograms_file = written_files["latency_histograms"]
```

```
    assert phase_metrics_file.exists()
    assert resource_usage_file.exists()
    assert latency_histograms_file.exists()

    assert phase_metrics_file.name == "phase_metrics.json"
    assert resource_usage_file.name == "resource_usage.jsonl"
    assert latency_histograms_file.name == "latency_histograms.json"


@pytest.mark.updated
def test_persist_all_metrics_creates_directory(
    tmp_path: Path,
    sample_full_metrics: dict[str, Any]
) -> None:
    """Test that persist_all_metrics creates output directory if needed."""
    nested_dir = tmp_path / "nested" / "metrics"
    assert not nested_dir.exists()

    written_files = persist_all_metrics(sample_full_metrics, nested_dir)

    assert nested_dir.exists()
    assert all(f.exists() for f in written_files.values())


@pytest.mark.updated
def test_persist_all_metrics_invalid_data(tmp_path: Path) -> None:
    """Test that invalid data raises ValueError."""
    with pytest.raises(ValueError, match="orchestrator_metrics must be a dictionary"):
        persist_all_metrics("not a dict", tmp_path)


@pytest.mark.updated
def test_validate_metrics_schema_valid(sample_full_metrics: dict[str, Any]) -> None:
    """Test validation of valid metrics schema."""
    errors = validate_metrics_schema(sample_full_metrics)
    assert errors == []


@pytest.mark.updated
def test_validate_metrics_schema_missing_keys() -> None:
    """Test validation detects missing required keys."""
    invalid_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {}
    }

    errors = validate_metrics_schema(invalid_metrics)

    assert len(errors) > 0
    assert any("resource_usage" in error for error in errors)
    assert any("abort_status" in error for error in errors)
    assert any("phase_status" in error for error in errors)
```

```
@pytest.mark.updated
def test_validate_metrics_schema_invalid_types() -> None:
    """Test validation detects invalid data types."""
    invalid_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": "not a dict",
        "resource_usage": "not a list",
        "abort_status": "not a dict",
        "phase_status": {}
    }

    errors = validate_metrics_schema(invalid_metrics)

    assert len(errors) >= 3
    assert any("phase_metrics must be a dictionary" in error for error in errors)
    assert any("resource_usage must be a list" in error for error in errors)
    assert any("abort_status must be a dictionary" in error for error in errors)


@pytest.mark.updated
def test_validate_metrics_schema_missing_phase_keys() -> None:
    """Test validation detects missing keys in phase metrics."""
    invalid_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": {
                "phase_id": 0,
                "name": "Test"
            }
        },
        "resource_usage": [],
        "abort_status": {},
        "phase_status": {}
    }

    errors = validate_metrics_schema(invalid_metrics)

    assert len(errors) > 0
      assert any("phase_metrics[0]" in error and "duration_ms" in error for error in
errors)
     assert any("phase_metrics[0]" in error and "items_processed" in error for error in
errors)
    assert any("phase_metrics[0]" in error and "latency_histogram" in error for error in
errors)


@pytest.mark.updated
def test_persist_metrics_idempotent(tmp_path: Path, sample_full_metrics: dict[str, Any])
-> None:
    """Test that persisting metrics twice produces identical results."""
    # First write
    written_files_1 = persist_all_metrics(sample_full_metrics, tmp_path)
```

```
    # Read back data
    with open(written_files_1["phase_metrics"], 'r') as f:
        data_1 = json.load(f)

    # Second write (should overwrite)
    written_files_2 = persist_all_metrics(sample_full_metrics, tmp_path)

    # Read back data again
    with open(written_files_2["phase_metrics"], 'r') as f:
        data_2 = json.load(f)

    # Should be identical
    assert data_1 == data_2
    assert written_files_1.keys() == written_files_2.keys()


@pytest.mark.updated
def test_persist_empty_resource_usage(tmp_path: Path) -> None:
    """Test persisting empty resource usage list."""
    output_file = persist_resource_usage([], tmp_path)

    assert output_file.exists()

    with open(output_file, 'r') as f:
        content = f.read()

    assert content == ""


@pytest.mark.updated
def test_json_serialization_deterministic(
    tmp_path: Path,
    sample_phase_metrics: dict[str, Any]
) -> None:
    """Test that JSON serialization is deterministic (sorted keys)."""
    output_file = persist_phase_metrics(sample_phase_metrics, tmp_path)

    with open(output_file, 'r') as f:
        content = f.read()

    # Parse and re-serialize to check key ordering
    data = json.loads(content)
    reserialized = json.dumps(data, indent=2, sort_keys=True, ensure_ascii=False)

    assert content == reserialized
```

```
tests/test_metrics_persistence_integration.py

"""Integration tests for orchestrator metrics persistence."""

import json
import pytest
from pathlib import Path
from typing import Any
from unittest.mock import Mock, MagicMock


@pytest.mark.updated
@pytest.mark.integration
def test_metrics_persistence_integration(tmp_path: Path) -> None:
    """Test end-to-end metrics persistence from orchestrator to files."""
    from farfan_pipeline.orchestration.orchestrator import (
        Orchestrator,
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    # Create a mock orchestrator with instrumentation
    orchestrator = Mock(spec=Orchestrator)

    # Create real ResourceLimits and PhaseInstrumentation
    resource_limits = ResourceLimits(
        max_memory_mb=2048.0,
        max_cpu_percent=80.0,
        max_workers=16
    )

    # Record some usage
    resource_limits.get_resource_usage()
    resource_limits.get_resource_usage()

    # Create phase instrumentation for phase 0
    phase0_instr = PhaseInstrumentation(
        phase_id=0,
        name="Configuration",
        items_total=1,
        resource_limits=resource_limits
    )
    phase0_instr.start()
    phase0_instr.increment(latency=0.123)
    phase0_instr.complete()

    # Create phase instrumentation for phase 1
    phase1_instr = PhaseInstrumentation(
        phase_id=1,
        name="Ingestion",
        items_total=1,
        resource_limits=resource_limits
    )
```

```
phase1_instr.start()
phase1_instr.increment(latency=0.456)
phase1_instr.complete()

# Mock export_metrics to return realistic data
def mock_export_metrics() -> dict[str, Any]:
    return {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": phase0_instr.build_metrics(),
            "1": phase1_instr.build_metrics()
        },
        "resource_usage": resource_limits.get_usage_history(),
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "0": "completed",
            "1": "completed"
        }
    }

orchestrator.export_metrics = mock_export_metrics

# Export and persist metrics
metrics = orchestrator.export_metrics()
written_files = persist_all_metrics(metrics, tmp_path)

# Verify all expected files were created
assert "phase_metrics" in written_files
assert "resource_usage" in written_files
assert "latency_histograms" in written_files

# Verify phase_metrics.json content
with open(written_files["phase_metrics"], 'r') as f:
    phase_metrics = json.load(f)

assert "0" in phase_metrics
assert "1" in phase_metrics
assert phase_metrics["0"]["phase_id"] == 0
assert phase_metrics["0"]["name"] == "Configuration"
assert phase_metrics["1"]["phase_id"] == 1
assert phase_metrics["1"]["name"] == "Ingestion"
assert phase_metrics["0"]["items_processed"] == 1
assert phase_metrics["1"]["items_processed"] == 1

# Verify resource_usage.jsonl content
with open(written_files["resource_usage"], 'r') as f:
    usage_lines = f.readlines()

assert len(usage_lines) >= 2
for line in usage_lines:
```

```
        entry = json.loads(line)
        assert "timestamp" in entry
        assert "cpu_percent" in entry
        assert "memory_percent" in entry
        assert "rss_mb" in entry
        assert "worker_budget" in entry

    # Verify latency_histograms.json content
    with open(written_files["latency_histograms"], 'r') as f:
        histograms = json.load(f)

    assert "0" in histograms
    assert "1" in histograms
    assert histograms["0"]["name"] == "Configuration"
    assert histograms["1"]["name"] == "Ingestion"
    assert "latency_histogram" in histograms["0"]
    assert "latency_histogram" in histograms["1"]


@pytest.mark.updated
@pytest.mark.integration
def test_metrics_match_in_memory_structures(tmp_path: Path) -> None:
    """Test that persisted metrics match in-memory structures."""
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    # Create instrumentation with specific data
    resource_limits = ResourceLimits(max_memory_mb=1024.0, max_workers=8)

    phase_instr = PhaseInstrumentation(
        phase_id=2,
        name="Micro Questions",
        items_total=300,
        resource_limits=resource_limits
    )

    phase_instr.start()
    for i in range(10):
        phase_instr.increment(latency=0.1 + i * 0.01)
    phase_instr.complete()

    # Build in-memory metrics
    in_memory_metrics = phase_instr.build_metrics()

    # Create full metrics structure
    full_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "2": in_memory_metrics
        },
        "resource_usage": resource_limits.get_usage_history(),
```

```
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "2": "completed"
        }
    }

    # Persist metrics
    written_files = persist_all_metrics(full_metrics, tmp_path)

    # Read back phase metrics
    with open(written_files["phase_metrics"], 'r') as f:
        persisted_metrics = json.load(f)

    # Compare in-memory vs persisted
    assert persisted_metrics["2"]["phase_id"] == in_memory_metrics["phase_id"]
    assert persisted_metrics["2"]["name"] == in_memory_metrics["name"]
                          assert      persisted_metrics["2"]["items_processed"]      ==
in_memory_metrics["items_processed"]
    assert persisted_metrics["2"]["items_total"] == in_memory_metrics["items_total"]
    assert persisted_metrics["2"]["duration_ms"] == in_memory_metrics["duration_ms"]
    assert persisted_metrics["2"]["throughput"] == in_memory_metrics["throughput"]
                         assert      persisted_metrics["2"]["latency_histogram"]      ==
in_memory_metrics["latency_histogram"]


@pytest.mark.updated
@pytest.mark.integration
def test_metrics_files_are_valid_json(tmp_path: Path) -> None:
    """Test that all persisted metrics files are valid JSON/JSONL."""
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    # Create minimal but valid metrics
    resource_limits = ResourceLimits()
    phase_instr = PhaseInstrumentation(phase_id=0, name="Test", items_total=1)
    phase_instr.start()
    phase_instr.increment()
    phase_instr.complete()

    full_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": phase_instr.build_metrics()
        },
        "resource_usage": resource_limits.get_usage_history(),
        "abort_status": {
            "is_aborted": False,
```

```
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "0": "completed"
        }
    }

    written_files = persist_all_metrics(full_metrics, tmp_path)

    # Validate phase_metrics.json
    with open(written_files["phase_metrics"], 'r') as f:
        try:
            json.load(f)
        except json.JSONDecodeError as e:
            pytest.fail(f"phase_metrics.json is not valid JSON: {e}")

    # Validate resource_usage.jsonl (each line must be valid JSON)
    with open(written_files["resource_usage"], 'r') as f:
        for line_num, line in enumerate(f, 1):
            if line.strip():
                try:
                    json.loads(line)
                except json.JSONDecodeError as e:
                        pytest.fail(f"resource_usage.jsonl line {line_num} is not valid
JSON: {e}")

    # Validate latency_histograms.json
    with open(written_files["latency_histograms"], 'r') as f:
        try:
            json.load(f)
        except json.JSONDecodeError as e:
            pytest.fail(f"latency_histograms.json is not valid JSON: {e}")


@pytest.mark.updated
@pytest.mark.integration
def test_metrics_persistence_handles_multiple_phases(tmp_path: Path) -> None:
    """Test metrics persistence with all 11 phases."""
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    resource_limits = ResourceLimits()

    # Create instrumentation for all 11 phases
    phase_names = [
        "Configuration", "Ingestion", "Micro Questions", "Scoring",
        "Dimensions", "Policy Areas", "Clusters", "Macro",
        "Recommendations", "Report", "Export"
    ]
```

```
    phase_metrics_dict = {}
    phase_status_dict = {}

    for phase_id, phase_name in enumerate(phase_names):
        phase_instr = PhaseInstrumentation(
            phase_id=phase_id,
            name=phase_name,
            items_total=10,
            resource_limits=resource_limits
        )
        phase_instr.start()
        for _ in range(10):
            phase_instr.increment(latency=0.1)
        phase_instr.complete()

        phase_metrics_dict[str(phase_id)] = phase_instr.build_metrics()
        phase_status_dict[str(phase_id)] = "completed"

    full_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": phase_metrics_dict,
        "resource_usage": resource_limits.get_usage_history(),
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": phase_status_dict
    }

    written_files = persist_all_metrics(full_metrics, tmp_path)

    # Verify all 11 phases are in phase_metrics.json
    with open(written_files["phase_metrics"], 'r') as f:
        persisted = json.load(f)

    assert len(persisted) == 11
    for i in range(11):
        assert str(i) in persisted
        assert persisted[str(i)]["phase_id"] == i
        assert persisted[str(i)]["name"] == phase_names[i]

    # Verify all 11 phases are in latency_histograms.json
    with open(written_files["latency_histograms"], 'r') as f:
        histograms = json.load(f)

    assert len(histograms) == 11
    for i in range(11):
        assert str(i) in histograms
        assert histograms[str(i)]["name"] == phase_names[i]


@pytest.mark.updated
@pytest.mark.integration
```

```
def test_metrics_persistence_with_warnings_and_errors(tmp_path: Path) -> None:
    """Test that warnings and errors are properly persisted."""
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    resource_limits = ResourceLimits()
    phase_instr = PhaseInstrumentation(
        phase_id=0,
        name="Test Phase",
        items_total=1,
        resource_limits=resource_limits
    )

    phase_instr.start()
    phase_instr.record_warning("integrity", "Test warning message", extra_field="value")
    phase_instr.record_error("execution", "Test error message", error_code=500)
    phase_instr.increment()
    phase_instr.complete()

    full_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": phase_instr.build_metrics()
        },
        "resource_usage": resource_limits.get_usage_history(),
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "0": "completed"
        }
    }

    written_files = persist_all_metrics(full_metrics, tmp_path)

    with open(written_files["phase_metrics"], 'r') as f:
        persisted = json.load(f)

    assert len(persisted["0"]["warnings"]) == 1
    assert persisted["0"]["warnings"][0]["category"] == "integrity"
    assert persisted["0"]["warnings"][0]["message"] == "Test warning message"
    assert persisted["0"]["warnings"][0]["extra_field"] == "value"

    assert len(persisted["0"]["errors"]) == 1
    assert persisted["0"]["errors"][0]["category"] == "execution"
    assert persisted["0"]["errors"][0]["message"] == "Test error message"
    assert persisted["0"]["errors"][0]["error_code"] == 500
```