tests/test_metrics_regression.py

```python
"""Regression tests to prevent metrics-only-in-logs issues."""

import json
import pytest
from pathlib import Path
from typing import Any
from unittest.mock import Mock, patch, MagicMock


@pytest.mark.updated
@pytest.mark.regression
def test_metrics_not_only_in_logs(tmp_path: Path) -> None:
    """
    Regression test: Ensure metrics are persisted to files, not just logged.

    This test guards against the original issue where export_metrics existed
    but was never called to write metrics to disk.
    """
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    # Create orchestrator-like metrics
    resource_limits = ResourceLimits()
    phase_instr = PhaseInstrumentation(phase_id=0, name="Test", items_total=1)
    phase_instr.start()
    phase_instr.increment()
    phase_instr.complete()

    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"
        }
    }

    # Persist metrics
    written_files = persist_all_metrics(metrics, tmp_path)

    # CRITICAL REGRESSION CHECK: Files must exist on disk
    assert written_files["phase_metrics"].exists(), \
        "REGRESSION: phase_metrics.json was not written to disk"
    assert written_files["resource_usage"].exists(), \
        "REGRESSION: resource_usage.jsonl was not written to disk"
    assert written_files["latency_histograms"].exists(), \
        "REGRESSION: latency_histograms.json was not written to disk"

    # Verify files have content (not empty)
    assert written_files["phase_metrics"].stat().st_size > 0, \
        "REGRESSION: phase_metrics.json is empty"
    assert written_files["latency_histograms"].stat().st_size > 0, \
        "REGRESSION: latency_histograms.json is empty"


@pytest.mark.updated
@pytest.mark.regression
def test_orchestrator_export_metrics_is_callable() -> None:
    """
    Regression test: Verify Orchestrator.export_metrics exists and is callable.

    Guards against accidental removal or renaming of the export_metrics method.
    """
    from farfan_pipeline.orchestration.orchestrator import Orchestrator

    # Verify method exists
    assert hasattr(Orchestrator, "export_metrics"), \
        "REGRESSION: Orchestrator.export_metrics method was removed"

    # Verify it's callable
    assert callable(getattr(Orchestrator, "export_metrics")), \
        "REGRESSION: Orchestrator.export_metrics is not callable"


@pytest.mark.updated
@pytest.mark.regression
def test_metrics_persistence_in_ci_artifacts_directory(tmp_path: Path) -> None:
    """
     Regression test: Ensure metrics are written to artifacts/ directory as expected by
CI.

    This test simulates the CI environment expectation that metrics files
    exist in artifacts/plan1/ after a pipeline run.
    """
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics

    # Simulate artifacts/plan1 directory structure
    artifacts_dir = tmp_path / "artifacts" / "plan1"
    artifacts_dir.mkdir(parents=True, exist_ok=True)

    # Create and persist metrics
    resource_limits = ResourceLimits()
```

```python
    phase_instr = PhaseInstrumentation(phase_id=0, name="Test", items_total=1)
    phase_instr.start()
    phase_instr.increment()
    phase_instr.complete()

    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(metrics, artifacts_dir)

    # CRITICAL: CI expects these exact file paths
    expected_phase_metrics = artifacts_dir / "phase_metrics.json"
    expected_resource_usage = artifacts_dir / "resource_usage.jsonl"
    expected_latency_histograms = artifacts_dir / "latency_histograms.json"

    assert expected_phase_metrics.exists(), \
        f"REGRESSION: CI cannot find {expected_phase_metrics}"
    assert expected_resource_usage.exists(), \
        f"REGRESSION: CI cannot find {expected_resource_usage}"
    assert expected_latency_histograms.exists(), \
        f"REGRESSION: CI cannot find {expected_latency_histograms}"


@pytest.mark.updated
@pytest.mark.regression
def test_metrics_files_have_required_content() -> None:
    """
    Regression test: Ensure metrics files contain the required fields for CI analysis.

    CI tools depend on specific fields being present in metrics files.
    """
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics
    from tempfile import TemporaryDirectory

    with TemporaryDirectory() as tmpdir:
        tmp_path = Path(tmpdir)
        resource_limits = ResourceLimits()
        phase_instr = PhaseInstrumentation(phase_id=0, name="Test", items_total=1)
        phase_instr.start()
        phase_instr.increment(latency=0.123)
        phase_instr.complete()

        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(metrics, tmp_path)

        # Check phase_metrics.json has required fields
        with open(written_files["phase_metrics"], 'r') as f:
            phase_data = json.load(f)

        assert "0" in phase_data, "REGRESSION: phase_metrics.json missing phase data"
        phase_0 = phase_data["0"]

        required_fields = [
            "phase_id", "name", "duration_ms", "items_processed",
            "items_total", "throughput", "latency_histogram"
        ]
        for field in required_fields:
            assert field in phase_0, \
                f"REGRESSION: phase_metrics.json missing required field: {field}"

        # Check latency_histograms.json has percentiles
        with open(written_files["latency_histograms"], 'r') as f:
            histogram_data = json.load(f)

        assert "0" in histogram_data, "REGRESSION: latency_histograms.json missing phase
data"
        assert "latency_histogram" in histogram_data["0"], \
            "REGRESSION: latency_histograms.json missing latency_histogram"

        hist = histogram_data["0"]["latency_histogram"]
        for percentile in ["p50", "p95", "p99"]:
            assert percentile in hist, \
                f"REGRESSION: latency_histograms.json missing {percentile}"
```

```python
@pytest.mark.updated
@pytest.mark.regression
def test_metrics_persistence_is_deterministic() -> None:
    """
    Regression test: Ensure metrics persistence is deterministic.

    Running the same metrics through persist_all_metrics twice
    should produce identical file contents.
    """
    from farfan_pipeline.orchestration.orchestrator import (
        PhaseInstrumentation,
        ResourceLimits
    )
    from farfan_pipeline.orchestration.metrics_persistence import persist_all_metrics
    from tempfile import TemporaryDirectory

    # Create fixed metrics data
    resource_limits = ResourceLimits()
    phase_instr = PhaseInstrumentation(phase_id=0, name="Test", items_total=1)
    phase_instr.start()
    phase_instr.increment(latency=0.123)
    phase_instr.complete()

    metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": phase_instr.build_metrics()
        },
        "resource_usage": [
            {
                "timestamp": "2024-01-01T00:00:00.000000",
                "cpu_percent": 50.0,
                "memory_percent": 30.0,
                "rss_mb": 512.0,
                "worker_budget": 8.0
            }
        ],
        "abort_status": {
            "is_aborted": False,
            "reason": None,
            "timestamp": None
        },
        "phase_status": {
            "0": "completed"
        }
    }

    # First write
    with TemporaryDirectory() as tmpdir1:
        tmp_path1 = Path(tmpdir1)
        written_files1 = persist_all_metrics(metrics, tmp_path1)

        with open(written_files1["phase_metrics"], 'r') as f:
            content1 = f.read()
    # Second write
    with TemporaryDirectory() as tmpdir2:
        tmp_path2 = Path(tmpdir2)
        written_files2 = persist_all_metrics(metrics, tmp_path2)

        with open(written_files2["phase_metrics"], 'r') as f:
            content2 = f.read()

    # Content should be byte-for-byte identical
    assert content1 == content2, \
        "REGRESSION: Metrics persistence is not deterministic"


@pytest.mark.updated
@pytest.mark.regression
def test_missing_metrics_files_detectable() -> None:
    """
    Regression test: Ensure missing metrics files can be detected by CI.

    If metrics persistence fails, CI should be able to detect the absence
    of expected files.
    """
    from pathlib import Path

    # Simulate a failed run where metrics weren't persisted
    fake_artifacts_dir = Path("/tmp/fake_artifacts_dir_that_does_not_exist")

    expected_files = [
        fake_artifacts_dir / "phase_metrics.json",
        fake_artifacts_dir / "resource_usage.jsonl",
        fake_artifacts_dir / "latency_histograms.json"
    ]

    # CI can detect missing files
    missing_files = [f for f in expected_files if not f.exists()]

    assert len(missing_files) == 3, \
        "REGRESSION: Cannot detect missing metrics files"

    # This test would fail in CI if metrics weren't persisted:
    # assert all(f.exists() for f in expected_files), \
    #     f"Missing metrics files: {missing_files}"


@pytest.mark.updated
@pytest.mark.regression
def test_metrics_schema_validation_prevents_invalid_data() -> None:
    """
      Regression test: Ensure schema validation prevents invalid metrics from being
accepted.

    This prevents silent failures where corrupted metrics are written to disk.
    """
```

```
                    from     farfan_pipeline.orchestration.metrics_persistence     import
validate_metrics_schema

    # Valid metrics should pass
    valid_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": {
            "0": {
                "phase_id": 0,
                "name": "Test",
                "duration_ms": 100.0,
                "items_processed": 1,
                "items_total": 1,
                "progress": 1.0,
                "throughput": 0.01,
                "warnings": [],
                "errors": [],
                "resource_snapshots": [],
                "latency_histogram": {"p50": 100.0, "p95": 100.0, "p99": 100.0},
                "anomalies": []
            }
        },
        "resource_usage": [],
        "abort_status": {},
        "phase_status": {}
    }

    errors = validate_metrics_schema(valid_metrics)
    assert len(errors) == 0, \
        f"REGRESSION: Valid metrics failed validation: {errors}"

    # Invalid metrics should fail
    invalid_metrics = {
        "timestamp": "2024-01-01T00:00:00.000000",
        "phase_metrics": "not a dict"
    }

    errors = validate_metrics_schema(invalid_metrics)
    assert len(errors) > 0, \
        "REGRESSION: Invalid metrics passed validation"
```

```
tests/test_model_output_verification.py

"""
Model Output Verification Tests
================================

Comprehensive test suite to verify that all Phase 0, Phase 1, and CPP models
can produce expected outputs and meet their contracts.

Tests:
- Phase 0 models instantiation and validation
- Phase 1 models instantiation for all 16 subphases
- CPP models instantiation and assembly
- Contract validation (invariants, pre/post conditions)
- Expected output structure verification

Author: F.A.R.F.A.N Testing Team
Date: 2025-12-10
"""

import pytest
from pathlib import Path
from datetime import datetime, timezone
from typing import List, Dict

# Phase 0 Models
from canonic_phases.Phase_one.phase0_input_validation import (
    Phase0Input,
    CanonicalInput,
    Phase0ValidationContract,
)

# Phase 1 Models
from canonic_phases.Phase_one.phase1_models import (
    LanguageData,
    PreprocessedDoc,
    StructureData,
    KnowledgeGraph,
    KGNode,
    KGEdge,
    Chunk,
    SmartChunk,
    CausalGraph,
    ValidationResult,
    CausalChains,
    IntegratedCausal,
    Arguments,
    Temporal,
    Discourse,
    Strategic,
)

# CPP Models
from canonic_phases.Phase_one.cpp_models import (
    CanonPolicyPackage,
    ChunkGraph,
    QualityMetrics,
    IntegrityIndex,
    PolicyManifest,
    LegacyChunk,
    TextSpan,
    ChunkResolution,
)


# ============================================================================
# PHASE 0 MODEL TESTS
# ============================================================================

class TestPhase0Models:
    """Test Phase 0 model instantiation and validation."""

    def test_phase0_input_instantiation(self):
        """Test Phase0Input can be instantiated with required fields."""
        input_data = Phase0Input(
            pdf_path=Path("/tmp/test.pdf"),
            run_id="test_run_001",
            questionnaire_path=None
        )

        assert input_data.pdf_path == Path("/tmp/test.pdf")
        assert input_data.run_id == "test_run_001"
        assert input_data.questionnaire_path is None

    def test_canonical_input_instantiation(self):
        """Test CanonicalInput can be instantiated with all required fields."""
        output = CanonicalInput(
            document_id="test_doc",
            run_id="test_run_001",
            pdf_path=Path("/tmp/test.pdf"),
            pdf_sha256="a" * 64,
            pdf_size_bytes=1000,
            pdf_page_count=10,
            questionnaire_path=Path("/tmp/q.json"),
            questionnaire_sha256="b" * 64,
            created_at=datetime.now(timezone.utc),
            phase0_version="1.0.0",
            validation_passed=True,
            validation_errors=[],
            validation_warnings=[]
        )

        # Verify required fields
        assert output.document_id == "test_doc"
        assert output.validation_passed is True
        assert len(output.validation_errors) == 0
        assert output.pdf_page_count == 10
        assert output.pdf_size_bytes == 1000
        assert len(output.pdf_sha256) == 64
        assert len(output.questionnaire_sha256) == 64

    def test_canonical_input_field_count(self):
        """Verify CanonicalInput has exactly 13 fields."""
        output = CanonicalInput(
            document_id="test",
            run_id="run",
            pdf_path=Path("/tmp/test.pdf"),
            pdf_sha256="a" * 64,
            pdf_size_bytes=100,
            pdf_page_count=1,
            questionnaire_path=Path("/tmp/q.json"),
            questionnaire_sha256="b" * 64,
            created_at=datetime.now(timezone.utc),
            phase0_version="1.0.0",
            validation_passed=True,
            validation_errors=[],
            validation_warnings=[]
        )

        assert len(output.__dataclass_fields__) == 13

    def test_phase0_validation_contract_instantiation(self):
        """Test Phase0ValidationContract can be instantiated."""
        contract = Phase0ValidationContract()

        assert contract.phase_name == "phase0_input_validation"
        assert len(contract.invariants) == 5

        # Verify invariant names
        invariant_names = {inv.name for inv in contract.invariants}
        expected_names = {
            "validation_passed",
            "pdf_page_count_positive",
            "pdf_size_positive",
            "sha256_format",
            "no_validation_errors"
        }
        assert invariant_names == expected_names


# ================================================================================
# PHASE 1 MODEL TESTS (SP0-SP15 Outputs)
# ================================================================================

class TestPhase1SubphaseModels:
    """Test Phase 1 subphase output models."""

    def test_sp0_language_data(self):
        """SP0: LanguageData output verification."""
        lang_data = LanguageData(
            primary_language="ES",
            secondary_languages=["EN"],
            confidence_scores={"ES": 0.95, "EN": 0.05},
            detection_method="langdetect"
        )

        assert lang_data.primary_language == "ES"
        assert len(lang_data.secondary_languages) >= 0
        assert "ES" in lang_data.confidence_scores

    def test_sp1_preprocessed_doc(self):
        """SP1: PreprocessedDoc output verification."""
        preprocessed = PreprocessedDoc(
            tokens=["palabra1", "palabra2", "palabra3"],
            sentences=["Sentencia uno.", "Sentencia dos."],
            paragraphs=["Párrafo uno."]
        )

        assert len(preprocessed.tokens) == 3
        assert len(preprocessed.sentences) == 2
        assert len(preprocessed.paragraphs) == 1

    def test_sp2_structure_data(self):
        """SP2: StructureData output verification."""
        structure = StructureData(
            sections=[{"id": "sec1", "title": "Section 1"}],
            hierarchy={"sec1": None},
            paragraph_mapping={0: "sec1"},
            unassigned_paragraphs=[]
        )

        assert len(structure.sections) >= 0
        assert isinstance(structure.hierarchy, dict)
        # Test alias property
        assert structure.paragraph_to_section == structure.paragraph_mapping

    def test_sp3_knowledge_graph(self):
        """SP3: KnowledgeGraph output verification."""
        node1 = KGNode(id="n1", type="entity", text="Entity 1")
        node2 = KGNode(id="n2", type="entity", text="Entity 2")
        edge = KGEdge(source="n1", target="n2", type="related", weight=1.0)

        kg = KnowledgeGraph(
            nodes=[node1, node2],
            edges=[edge]
        )

        assert len(kg.nodes) == 2
        assert len(kg.edges) == 1
        assert kg.nodes[0].id == "n1"
        assert kg.edges[0].source == "n1"

    def test_sp4_chunk_output(self):
        """SP4: Chunk output verification (60 chunks)."""
        chunks: List[Chunk] = []
        # Generate 60 chunks (PA01-PA10 × DIM01-DIM06)
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                chunk = Chunk(
                    chunk_id=f"PA{pa_num:02d}-DIM{dim_num:02d}",
                    policy_area_id=f"PA{pa_num:02d}",
                    dimension_id=f"DIM{dim_num:02d}",
                    chunk_index=(pa_num - 1) * 6 + (dim_num - 1),
                    text=f"Content for PA{pa_num:02d}-DIM{dim_num:02d}"
                )
                chunks.append(chunk)

        # Verify 60 chunks (CONSTITUTIONAL INVARIANT)
        assert len(chunks) == 60

        # Verify unique chunk_ids
        chunk_ids = {c.chunk_id for c in chunks}
        assert len(chunk_ids) == 60

        # Verify PA×DIM coverage
        expected_ids = {
            f"PA{pa:02d}-DIM{dim:02d}"
            for pa in range(1, 11)
            for dim in range(1, 7)
        }
        assert chunk_ids == expected_ids

    def test_sp5_causal_chains(self):
        """SP5: CausalChains output verification."""
        causal = CausalChains(
            chains=[{"cause": "A", "effect": "B"}],
            mechanisms=["mechanism1"],
            per_chunk_causal={"PA01-DIM01": {"chains": 1}}
        )

        assert len(causal.chains) >= 0
        assert isinstance(causal.per_chunk_causal, dict)

    def test_sp6_integrated_causal(self):
        """SP6: IntegratedCausal output verification."""
        integrated = IntegratedCausal(
            global_graph={"nodes": [], "edges": []},
            validated_hierarchy=True,
            cross_chunk_links=[],
            teoria_cambio_status="OK"
        )

        assert isinstance(integrated.global_graph, dict)
        assert isinstance(integrated.validated_hierarchy, bool)

    def test_sp7_arguments(self):
        """SP7: Arguments output verification."""
        args = Arguments(
            premises=[{"id": "p1", "text": "Premise 1"}],
            conclusions=[{"id": "c1", "text": "Conclusion 1"}],
            reasoning=[{"pattern": "deductive"}],
            per_chunk_args={}
        )

        assert len(args.premises) >= 0
        assert len(args.conclusions) >= 0

    def test_sp8_temporal(self):
        """SP8: Temporal output verification."""
        temporal = Temporal(
            time_markers=[{"date": "2025-01-01"}],
            sequences=[{"start": "A", "end": "B"}],
            durations=[{"duration": "1 year"}],
            per_chunk_temporal={}
        )

        assert len(temporal.time_markers) >= 0
        assert isinstance(temporal.per_chunk_temporal, dict)

    def test_sp9_discourse(self):
        """SP9: Discourse output verification."""
        discourse = Discourse(
            markers=[{"type": "connector", "text": "therefore"}],
            patterns=[{"pattern": "argumentative"}],
            coherence={"score": 0.8},
            per_chunk_discourse={}
        )

        assert len(discourse.markers) >= 0
        assert "score" in discourse.coherence or len(discourse.coherence) == 0

    def test_sp10_strategic(self):
        """SP10: Strategic output verification."""
        strategic = Strategic(
            strategic_rank={"PA01-DIM01": 75},
            priorities=[{"rank": 1, "chunk_id": "PA01-DIM01"}],
            integrated_view={},
            strategic_scores={}
        )

        assert isinstance(strategic.strategic_rank, dict)
        # Verify ranks in valid range [0, 100]
        for rank in strategic.strategic_rank.values():
            assert 0 <= rank <= 100

    def test_sp11_smart_chunk_output(self):
        """SP11: SmartChunk output verification (60 smart chunks)."""
        smart_chunks: List[SmartChunk] = []

        # Generate 60 smart chunks
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                chunk_id = f"PA{pa_num:02d}-DIM{dim_num:02d}"
                smart_chunk = SmartChunk(
                    chunk_id=chunk_id,
                    text=f"Content for {chunk_id}",
                    chunk_type="MACRO",
                    chunk_index=(pa_num - 1) * 6 + (dim_num - 1),
                    policy_area_id=f"PA{pa_num:02d}",
                    dimension_id=f"DIM{dim_num:02d}",
                    causal_graph=CausalGraph(),
                    temporal_markers={},
                    arguments={},
                    discourse_mode="default",
                    strategic_rank=50,
                    irrigation_links=[],
                    signal_tags=[],
                    signal_scores={},
                    signal_version="1.0",
                    rank_score=0.5,
                    signal_weighted_score=0.5
                )
                smart_chunks.append(smart_chunk)

        # Verify 60 smart chunks (CONSTITUTIONAL INVARIANT)
        assert len(smart_chunks) == 60

        # Verify all have strategic_rank
        for sc in smart_chunks:
            assert hasattr(sc, 'strategic_rank')
            assert isinstance(sc.strategic_rank, (int, float))

    def test_sp13_validation_result(self):
        """SP13: ValidationResult output verification."""
        validation = ValidationResult(
            status="VALID",
            violations=[],
            checked_count=60,
            passed_count=60
        )

        assert validation.status in ["VALID", "INVALID"]
        assert validation.passed_count <= validation.checked_count
        assert len(validation.violations) == 0  # For VALID status


# ==============================================================================
# CPP MODEL TESTS
# ==============================================================================

class TestCPPModels:
    """Test CPP (CanonPolicyPackage) model instantiation."""

    def test_text_span(self):
        """Test TextSpan model."""
        span = TextSpan(start=0, end=100)
        assert span.start == 0
        assert span.end == 100
        assert span.start <= span.end

    def test_legacy_chunk(self):
        """Test LegacyChunk model with validation."""
        chunk = LegacyChunk(
            id="PA01_DIM01",
            text="Test content",
            text_span=TextSpan(0, 100),
            resolution=ChunkResolution.MACRO,
            bytes_hash="abc123def456",
            policy_area_id="PA01",
            dimension_id="DIM01"
        )

        assert chunk.id == "PA01_DIM01"
        assert chunk.policy_area_id == "PA01"
        assert chunk.dimension_id == "DIM01"
        assert chunk.resolution == ChunkResolution.MACRO

    def test_legacy_chunk_validation_invalid_pa(self):
        """Test LegacyChunk rejects invalid policy_area_id."""
        with pytest.raises(ValueError, match="Invalid policy_area_id"):
            LegacyChunk(
                id="PA99_DIM01",
                text="Test",
                text_span=TextSpan(0, 10),
                resolution=ChunkResolution.MACRO,
                bytes_hash="abc",
                policy_area_id="PA99",  # Invalid
                dimension_id="DIM01"
            )

    def test_legacy_chunk_validation_invalid_dim(self):
        """Test LegacyChunk rejects invalid dimension_id."""
        with pytest.raises(ValueError, match="Invalid dimension_id"):
            LegacyChunk(
                id="PA01_DIM99",
                text="Test",
                text_span=TextSpan(0, 10),
                resolution=ChunkResolution.MACRO,
                bytes_hash="abc",
                policy_area_id="PA01",
                dimension_id="DIM99"  # Invalid
            )

    def test_chunk_graph_60_chunks(self):
        """Test ChunkGraph with 60 chunks."""
        chunk_graph = ChunkGraph()

        # Add 60 chunks
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                chunk_id = f"PA{pa_num:02d}_DIM{dim_num:02d}"
                chunk = LegacyChunk(
                    id=chunk_id,
                    text=f"Content for {chunk_id}",
                    text_span=TextSpan(0, 100),
                    resolution=ChunkResolution.MACRO,
                    bytes_hash="abc123",
                    policy_area_id=f"PA{pa_num:02d}",
                    dimension_id=f"DIM{dim_num:02d}"
                )
                chunk_graph.chunks[chunk_id] = chunk

        # Verify 60 chunks
        assert len(chunk_graph.chunks) == 60
        assert chunk_graph.chunk_count == 60

    def test_quality_metrics(self):
        """Test QualityMetrics model."""
        quality = QualityMetrics(
            provenance_completeness=0.85,
            structural_consistency=0.90,
            chunk_count=60,
            coverage_analysis={},
            signal_quality_by_pa={}
        )

        # Verify thresholds per FORCING ROUTE
        assert quality.provenance_completeness >= 0.8  # [POST-002]
        assert quality.structural_consistency >= 0.85  # [POST-003]
        assert quality.chunk_count == 60

    def test_integrity_index(self):
        """Test IntegrityIndex model."""
        integrity = IntegrityIndex(
            blake2b_root="a" * 64,
            chunk_hashes=("chunk1hash", "chunk2hash"),
            timestamp="2025-12-10T17:00:00Z"
        )

        assert len(integrity.blake2b_root) > 0
        assert isinstance(integrity.timestamp, str)

    def test_policy_manifest(self):
        """Test PolicyManifest model."""
        manifest = PolicyManifest(
            questionnaire_version="1.0.0",
            questionnaire_sha256="a" * 64,
            policy_areas=tuple(f"PA{i:02d}" for i in range(1, 11)),
            dimensions=tuple(f"DIM{i:02d}" for i in range(1, 7))
        )

        assert len(manifest.policy_areas) == 10
        assert len(manifest.dimensions) == 6
        assert manifest.questionnaire_version == "1.0.0"
```

```python
def test_canon_policy_package_assembly(self):
    """Test CanonPolicyPackage full assembly."""
    # Build ChunkGraph with 1 chunk (for testing)
    chunk_graph = ChunkGraph()
    chunk = LegacyChunk(
        id="PA01_DIM01",
        text="Test content",
        text_span=TextSpan(0, 100),
        resolution=ChunkResolution.MACRO,
        bytes_hash="abc123",
        policy_area_id="PA01",
        dimension_id="DIM01"
    )
    chunk_graph.chunks[chunk.id] = chunk

    # Build QualityMetrics
    quality = QualityMetrics(
        provenance_completeness=0.85,
        structural_consistency=0.90,
        chunk_count=1,  # For test
        coverage_analysis={},
        signal_quality_by_pa={}
    )

    # Build IntegrityIndex
    integrity = IntegrityIndex(
        blake2b_root="a" * 64,
        chunk_hashes=("abc123",),
        timestamp="2025-12-10T17:00:00Z"
    )

    # Build PolicyManifest
    manifest = PolicyManifest(
        questionnaire_version="1.0.0",
        questionnaire_sha256="a" * 64,
        policy_areas=("PA01",),
        dimensions=("DIM01",)
    )

    # Assemble CPP
    cpp = CanonPolicyPackage(
        schema_version="SPC-2025.1",
        document_id="test_doc",
        chunk_graph=chunk_graph,
        quality_metrics=quality,
        integrity_index=integrity,
        policy_manifest=manifest,
        metadata={}
    )

    # Verify CPP structure
    assert cpp.schema_version == "SPC-2025.1"
    assert cpp.document_id == "test_doc"
```

```
        assert len(cpp.chunk_graph.chunks) >= 1
        assert cpp.quality_metrics.provenance_completeness >= 0.8
        assert cpp.quality_metrics.structural_consistency >= 0.85


# ================================================================================
# CONSTITUTIONAL INVARIANT TESTS
# ================================================================================

class TestConstitutionalInvariants:
    """Test that constitutional invariants are enforced."""

    def test_60_chunk_invariant_sp4(self):
        """Verify SP4 produces exactly 60 chunks."""
        chunks: List[Chunk] = []

        for pa in range(1, 11):
            for dim in range(1, 7):
                chunk = Chunk(
                    chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                    policy_area_id=f"PA{pa:02d}",
                    dimension_id=f"DIM{dim:02d}",
                    text="content"
                )
                chunks.append(chunk)

        # CONSTITUTIONAL INVARIANT
              assert len(chunks) == 60, f"SP4 must produce exactly 60 chunks, got
{len(chunks)}"

    def test_60_chunk_invariant_sp11(self):
        """Verify SP11 produces exactly 60 smart chunks."""
        smart_chunks: List[SmartChunk] = []

        for pa in range(1, 11):
            for dim in range(1, 7):
                sc = SmartChunk(
                    chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                    text="content",
                    chunk_type="MACRO",
                    chunk_index=0,
                    policy_area_id=f"PA{pa:02d}",
                    dimension_id=f"DIM{dim:02d}",
                    causal_graph=CausalGraph(),
                    temporal_markers={},
                    arguments={},
                    discourse_mode="default",
                    strategic_rank=50,
                    irrigation_links=[],
                    signal_tags=[],
                    signal_scores={},
                    signal_version="1.0",
                    rank_score=0.5,
                    signal_weighted_score=0.5
                )
                smart_chunks.append(sc)

        # CONSTITUTIONAL INVARIANT
        assert len(smart_chunks) == 60, f"SP11 must produce exactly 60 smart chunks, got
{len(smart_chunks)}"

    def test_60_chunk_invariant_cpp(self):
        """Verify CPP ChunkGraph has exactly 60 chunks."""
        chunk_graph = ChunkGraph()

        for pa in range(1, 11):
            for dim in range(1, 7):
                chunk_id = f"PA{pa:02d}_DIM{dim:02d}"
                chunk = LegacyChunk(
                    id=chunk_id,
                    text="content",
                    text_span=TextSpan(0, 10),
                    resolution=ChunkResolution.MACRO,
                    bytes_hash="abc",
                    policy_area_id=f"PA{pa:02d}",
                    dimension_id=f"DIM{dim:02d}"
                )
                chunk_graph.chunks[chunk_id] = chunk

        # CONSTITUTIONAL INVARIANT
        assert chunk_graph.chunk_count == 60, f"CPP must have exactly 60 chunks, got
{chunk_graph.chunk_count}"

    def test_padim_coverage_complete(self):
        """Verify PA×DIM grid coverage is complete (all 60 cells)."""
        expected_ids = {
            f"PA{pa:02d}-DIM{dim:02d}"
            for pa in range(1, 11)
            for dim in range(1, 7)
        }

        assert len(expected_ids) == 60

        # Verify format
        for chunk_id in expected_ids:
            assert chunk_id.startswith("PA")
            assert "-DIM" in chunk_id


# ==============================================================================
# RUN TESTS
# ==============================================================================

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

tests/test_nexus_scoring_alignment.py

```
"""
Tests for Nexus-Scoring Alignment and Interface Stability
=========================================================

This test suite verifies the harmonization between Phase 2 (EvidenceNexus)
and Phase 3 (Scoring), ensuring stable entry point for scoring operations.

Test Categories:
1. Interface Contract Tests
2. Evidence Structure Tests
3. Scoring Modality Tests
4. Adaptive Threshold Tests
5. End-to-End Integration Tests

Author: F.A.R.F.A.N Pipeline Team
Version: 1.0.0
Date: 2025-12-11
"""

import pytest
from typing import Any

from farfan_pipeline.analysis.scoring.scoring import (
    EvidenceStructureError,
    ModalityConfig,
    ModalityValidationError,
    QualityLevel,
    ScoredResult,
    ScoringValidator,
    apply_scoring,
    determine_quality_level,
)
from farfan_pipeline.analysis.scoring.nexus_scoring_validator import (
    NexusScoringValidator,
    BatchValidator,
    ValidationResult,
)


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

@pytest.fixture
def valid_nexus_evidence() -> dict[str, Any]:
    """Valid evidence structure from Phase 2."""
    return {
        "elements": [
            {"text": "Element 1", "source": "doc1", "confidence": 0.9},
            {"text": "Element 2", "source": "doc2", "confidence": 0.8},
            {"text": "Element 3", "source": "doc3", "confidence": 0.85},
        ],
        "by_type": {
            "indicator_numeric": [{"value": 100, "unit": "personas"}],
            "territorial_coverage": [{"region": "nacional"}],
        },
        "confidence": 0.85,
        "completeness": 0.90,
        "graph_hash": "a" * 64,  # Valid SHA-256 hex length
        "patterns": {
            "pattern1": ["match1", "match2"],
            "pattern2": ["match3"],
        }
    }


@pytest.fixture
def valid_micro_question_run(valid_nexus_evidence: dict[str, Any]) -> dict[str, Any]:
    """Valid MicroQuestionRun from Phase 2."""
    return {
        "question_id": "Q001",
        "question_global": 1,
        "base_slot": "D1-Q1",
        "metadata": {
            "policy_area_id": "PA01",
            "dimension_id": "DIM01",
        },
        "evidence": valid_nexus_evidence,
        "error": None,
        "duration_ms": 123.45,
        "aborted": False,
    }


@pytest.fixture
def valid_scoring_context() -> dict[str, Any]:
    """Valid scoring context from SISAS."""
    return {
        "modality": "TYPE_A",
        "threshold": 0.75,
        "weight_elements": 0.5,
        "weight_similarity": 0.3,
        "weight_patterns": 0.2,
        "aggregation_method": "weighted_mean",
    }


# ============================================================================
# INTERFACE CONTRACT TESTS
# ============================================================================

class TestInterfaceContract:
    """Test interface contract between Nexus and Scoring."""

    def test_valid_nexus_output_structure(
        self,
        valid_micro_question_run: dict[str, Any]
    ) -> None:
        """Test valid nexus output passes validation."""
        result = NexusScoringValidator.validate_nexus_output(valid_micro_question_run)

        assert result.is_valid
        assert len(result.errors) == 0
        assert result.metadata["has_evidence"]
        assert result.metadata["confidence"] == 0.85

    def test_missing_evidence_key(self) -> None:
        """Test missing evidence key fails validation."""
        invalid_run = {
            "question_id": "Q001",
            "metadata": {},
        }

        result = NexusScoringValidator.validate_nexus_output(invalid_run)

        assert not result.is_valid
        assert any("evidence" in err for err in result.errors)

    def test_none_evidence_handled_gracefully(self) -> None:
        """Test None evidence (failed question) handled gracefully."""
        run_with_none_evidence = {
            "question_id": "Q001",
            "evidence": None,
            "error": "Extraction failed",
        }

        result = NexusScoringValidator.validate_nexus_output(run_with_none_evidence)

        assert result.is_valid  # None is valid for failed questions
        assert len(result.warnings) > 0
        assert not result.metadata["has_evidence"]

    def test_missing_required_evidence_keys(self) -> None:
        """Test missing required evidence keys."""
        incomplete_evidence = {
            "elements": [],
            # Missing "confidence"
        }
        run = {"evidence": incomplete_evidence}

        result = NexusScoringValidator.validate_nexus_output(run)

        assert not result.is_valid
        assert any("confidence" in err for err in result.errors)


# ============================================================================
# EVIDENCE STRUCTURE TESTS
# ============================================================================
```

```python
class TestEvidenceStructure:
    """Test evidence structure validation."""

    def test_scoring_validator_accepts_valid_evidence(
        self,
        valid_nexus_evidence: dict[str, Any]
    ) -> None:
        """Test ScoringValidator accepts valid nexus evidence."""
        # Should not raise exception
        ScoringValidator.validate_evidence(valid_nexus_evidence)

    def test_scoring_validator_rejects_invalid_type(self) -> None:
        """Test ScoringValidator rejects non-dict evidence."""
        with pytest.raises(EvidenceStructureError, match="must be dict"):
            ScoringValidator.validate_evidence("invalid")

    def test_scoring_validator_rejects_missing_elements(self) -> None:
        """Test ScoringValidator rejects evidence without elements."""
        invalid = {"confidence": 0.5}  # Missing "elements"

        with pytest.raises(EvidenceStructureError, match="elements"):
            ScoringValidator.validate_evidence(invalid)

    def test_scoring_validator_rejects_invalid_confidence(self) -> None:
        """Test ScoringValidator rejects out-of-range confidence."""
        invalid = {
            "elements": [],
            "confidence": 1.5  # Out of range
        }

        with pytest.raises(EvidenceStructureError, match="confidence"):
            ScoringValidator.validate_evidence(invalid)

    def test_extract_scores_from_evidence(
        self,
        valid_nexus_evidence: dict[str, Any]
    ) -> None:
        """Test score extraction from evidence."""
        scores = ScoringValidator.extract_scores(valid_nexus_evidence)

        assert "elements_score" in scores
        assert "similarity_score" in scores
        assert "patterns_score" in scores

        # All scores in [0, 1]
        for score in scores.values():
            assert 0.0 <= score <= 1.0


# ============================================================================
# SCORING MODALITY TESTS
# ============================================================================

class TestScoringModalities:
    """Test scoring modality functions."""

        @pytest.mark.parametrize("modality",  ["TYPE_A",  "TYPE_B",  "TYPE_C",  "TYPE_D",
"TYPE_E", "TYPE_F"])
    def test_all_modalities_produce_valid_scores(
        self,
        valid_nexus_evidence: dict[str, Any],
        modality: str
    ) -> None:
        """Test all modalities produce valid scores."""
        result = apply_scoring(valid_nexus_evidence, modality)  # type: ignore

        assert isinstance(result, ScoredResult)
        assert 0.0 <= result.score <= 1.0
        assert 0.0 <= result.normalized_score <= 100.0
        assert isinstance(result.quality_level, QualityLevel)
        assert isinstance(result.passes_threshold, bool)
        assert len(result.confidence_interval) == 2

    def test_type_a_high_threshold(
        self,
        valid_nexus_evidence: dict[str, Any]
    ) -> None:
        """Test TYPE_A uses high threshold (0.75)."""
        result = apply_scoring(valid_nexus_evidence, "TYPE_A")

        assert result.scoring_metadata["threshold"] == 0.75
        assert result.scoring_metadata["modality"] == "TYPE_A"

    def test_type_b_pattern_emphasis(
        self,
        valid_nexus_evidence: dict[str, Any]
    ) -> None:
        """Test TYPE_B emphasizes patterns."""
        result = apply_scoring(valid_nexus_evidence, "TYPE_B")

        # TYPE_B should have higher pattern weight
        assert result.scoring_metadata["threshold"] == 0.65

    def test_modality_config_validation(self) -> None:
        """Test ModalityConfig validation."""
        # Valid config
        config = ModalityConfig(
            modality="TYPE_A",
            threshold=0.75,
            weight_elements=0.5,
            weight_similarity=0.3,
            weight_patterns=0.2
        )
        assert config.threshold == 0.75

        # Invalid threshold
        with pytest.raises(ModalityValidationError):
            ModalityConfig(
                modality="TYPE_A",
                threshold=1.5,  # Out of range
                weight_elements=0.5,
                weight_similarity=0.3,
                weight_patterns=0.2
            )

        # Invalid weights (don't sum to 1)
        with pytest.raises(ModalityValidationError):
            ModalityConfig(
                modality="TYPE_A",
                threshold=0.75,
                weight_elements=0.5,
                weight_similarity=0.5,
                weight_patterns=0.5  # Sum > 1
            )


# ============================================================================
# ADAPTIVE THRESHOLD TESTS
# ============================================================================

class TestAdaptiveThresholds:
    """Test adaptive threshold computation and application."""

    def test_scoring_context_validation(
        self,
        valid_scoring_context: dict[str, Any]
    ) -> None:
        """Test scoring context validation."""
        result = NexusScoringValidator.validate_scoring_context(valid_scoring_context)

        assert result.is_valid
        assert result.metadata["has_context"]
        assert result.metadata["modality"] == "TYPE_A"
        assert result.metadata["threshold"] == 0.75

    def test_missing_scoring_context_handled(self) -> None:
        """Test missing scoring context uses defaults."""
        result = NexusScoringValidator.validate_scoring_context(None)

        assert result.is_valid
        assert not result.metadata["has_context"]
        assert len(result.warnings) > 0

    def test_invalid_threshold_rejected(self) -> None:
        """Test invalid threshold rejected."""
        invalid_context = {
            "modality": "TYPE_A",
            "threshold": 1.5  # Out of range
        }

        result = NexusScoringValidator.validate_scoring_context(invalid_context)
```

```python
        assert not result.is_valid
        assert any("threshold" in err.lower() for err in result.errors)


# ============================================================================
# QUALITY LEVEL TESTS
# ============================================================================

class TestQualityLevels:
    """Test quality level determination."""

    @pytest.mark.parametrize("score,expected_level", [
        (0.90, QualityLevel.EXCELLENT),
        (0.85, QualityLevel.EXCELLENT),
        (0.75, QualityLevel.GOOD),
        (0.70, QualityLevel.GOOD),
        (0.60, QualityLevel.ADEQUATE),
        (0.50, QualityLevel.ADEQUATE),
        (0.40, QualityLevel.POOR),
        (0.20, QualityLevel.POOR),
    ])
    def test_quality_level_thresholds(
        self,
        score: float,
        expected_level: QualityLevel
    ) -> None:
        """Test quality level determination matches thresholds."""
        level = determine_quality_level(score)
        assert level == expected_level


# ============================================================================
# PHASE TRANSITION TESTS
# ============================================================================

class TestPhaseTransition:
    """Test complete Phase 2 ? Phase 3 transition."""

    def test_valid_phase_transition(
        self,
        valid_micro_question_run: dict[str, Any],
        valid_scoring_context: dict[str, Any]
    ) -> None:
        """Test valid phase transition."""
        result = NexusScoringValidator.validate_phase_transition(
            valid_micro_question_run,
            valid_scoring_context
        )

        assert result.is_valid
        assert result.metadata["overall_valid"]
        assert result.metadata["nexus_validation"]["has_evidence"]
        assert result.metadata["context_validation"]["has_context"]
    def test_confidence_threshold_alignment(
        self,
        valid_micro_question_run: dict[str, Any]
    ) -> None:
        """Test confidence vs threshold alignment warning."""
        # Set low threshold
        low_threshold_context = {
            "modality": "TYPE_A",
            "threshold": 0.90,  # Higher than evidence confidence (0.85)
        }

        result = NexusScoringValidator.validate_phase_transition(
            valid_micro_question_run,
            low_threshold_context
        )

        # Should be valid but with warning
        assert result.is_valid
        assert len(result.warnings) > 0
        assert "confidence_threshold_delta" in result.metadata


# ===============================================================================
# BATCH VALIDATION TESTS
# ===============================================================================

class TestBatchValidation:
    """Test batch validation for multiple questions."""

    def test_batch_validation_all_valid(
        self,
        valid_micro_question_run: dict[str, Any],
        valid_scoring_context: dict[str, Any]
    ) -> None:
        """Test batch validation with all valid runs."""
        runs = [valid_micro_question_run.copy() for _ in range(10)]
        contexts = [valid_scoring_context.copy() for _ in range(10)]

        batch_result = BatchValidator.validate_batch(runs, contexts)

        assert batch_result["total_validations"] == 10
        assert batch_result["valid_count"] == 10
        assert batch_result["success_rate"] == 1.0
        assert batch_result["total_errors"] == 0

    def test_batch_validation_mixed_results(
        self,
        valid_micro_question_run: dict[str, Any]
    ) -> None:
        """Test batch validation with mixed results."""
        # Create mix of valid and invalid runs
        runs = []
        for i in range(5):
            runs.append(valid_micro_question_run.copy())
        for i in range(5):
            invalid_run = {"question_id": f"Q{i:03d}"}  # Missing evidence
            runs.append(invalid_run)

        batch_result = BatchValidator.validate_batch(runs)

        assert batch_result["total_validations"] == 10
        assert batch_result["valid_count"] == 5
        assert batch_result["invalid_count"] == 5
        assert batch_result["success_rate"] == 0.5


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

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

    def test_complete_nexus_to_scoring_flow(
        self,
        valid_nexus_evidence: dict[str, Any]
    ) -> None:
        """Test complete flow from nexus evidence to scored result."""
        # 1. Validate evidence structure
        ScoringValidator.validate_evidence(valid_nexus_evidence)

        # 2. Apply scoring
        scored_result = apply_scoring(valid_nexus_evidence, "TYPE_A")

        # 3. Verify result structure
        assert isinstance(scored_result, ScoredResult)
        assert scored_result.score > 0
        assert scored_result.quality_level in QualityLevel

        # 4. Convert to dict (for serialization)
        result_dict = scored_result.to_dict()
        assert "score" in result_dict
        assert "quality_level" in result_dict

    def test_300_questions_interface_stability(
        self,
        valid_micro_question_run: dict[str, Any],
        valid_scoring_context: dict[str, Any]
    ) -> None:
        """Test interface stability for all 300 questions."""
        # Simulate 300 questions
        runs = []
        contexts = []

        for i in range(1, 301):
            run = valid_micro_question_run.copy()
            run["question_id"] = f"Q{i:03d}"
            run["question_global"] = i
            runs.append(run)

            context = valid_scoring_context.copy()
            # Vary modalities
            modalities = ["TYPE_A", "TYPE_B", "TYPE_C", "TYPE_D", "TYPE_E", "TYPE_F"]
            context["modality"] = modalities[i % len(modalities)]
            contexts.append(context)

        # Batch validate
        batch_result = BatchValidator.validate_batch(runs, contexts)

        # All should be valid
        assert batch_result["total_validations"] == 300
        assert batch_result["success_rate"] == 1.0

        print(f"\n? Interface stability confirmed for 300 questions")
        print(f"   Valid: {batch_result['valid_count']}/300")
        print(f"   Success rate: {batch_result['success_rate']:.2%}")


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

pytestmark = pytest.mark.updated
```

tests/test_orchestrator_mode_integration.py

```python
"""
Integration Test for Orchestrator Mode-Specific Behaviors

This test suite validates that the orchestrator correctly enforces runtime mode
behaviors throughout the pipeline execution, particularly in Phase 0 configuration
loading.

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

import hashlib
import json
import os
import sys
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 (
    RuntimeConfig,
    RuntimeMode,
    reset_runtime_config,
)


def _normalize_monolith_for_hash_standalone(monolith: dict | MappingProxyType) -> dict:
    """
    Standalone copy of normalization logic for testing without heavy imports.

    This is identical to orchestrator._normalize_monolith_for_hash but extracted
    for testing purposes to avoid importing the entire orchestrator module.
    """
    if isinstance(monolith, MappingProxyType):
        monolith = dict(monolith)

    def _convert(obj: Any) -> Any:
        """Recursively convert proxy types to canonical forms."""
        if isinstance(obj, MappingProxyType):
            obj = dict(obj)
        if isinstance(obj, dict):
            return {k: _convert(v) for k, v in obj.items()}
        if isinstance(obj, list):
            return [_convert(v) for v in obj]
        return obj

    normalized = _convert(monolith)
    try:
            json.dumps(normalized, sort_keys=True, ensure_ascii=False, separators=(",",
":"))
    except (TypeError, ValueError) as exc:
        raise RuntimeError(
            f"Monolith normalization failed: contains non-serializable types. "
            f"All monolith content must be JSON-serializable. Error: {exc}"
        ) from exc

    return normalized


class TestOrchestratorModeIntegration:
    """Integration tests for orchestrator mode-specific behaviors."""

    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_sets_verified_status(self):
        """Test PROD mode sets verification_status to 'verified' in config dict."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        mock_monolith = {
            "blocks": {
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 306)],
                "meso_questions": [],
                "macro_question": {},
            },
            "version": "1.0.0",
        }

        normalized = _normalize_monolith_for_hash_standalone(mock_monolith)
        assert normalized is not None
        assert "blocks" in normalized

        print("  ? PROD mode normalizes monolith correctly")

    def test_dev_mode_marks_development_output(self):
        """Test DEV mode marks output as 'development'."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
        config = RuntimeConfig.from_env()

        assert config.mode == RuntimeMode.DEV
        assert config.allow_aggregation_defaults

        print("  ? DEV mode allows aggregation defaults")

    def test_exploratory_mode_marks_experimental_output(self):
        """Test EXPLORATORY mode marks output as '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_fails_on_question_count_mismatch(self):
        """Test PROD mode with strict enforcement fails on question count mismatch."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        assert config.mode == RuntimeMode.PROD
        assert config.is_strict_mode()

        mock_monolith = {
            "blocks": {
                "micro_questions": [{"id": "Q001"}],  # Only 1, not 305
                "meso_questions": [],
                "macro_question": {},
            }
        }

        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        normalized = _normalize_monolith_for_hash(mock_monolith)
        assert normalized is not None

        print("  ? PROD mode normalizes monolith even with count mismatch")

    def test_hash_normalization_with_proxy_types(self):
        """Test hash normalization handles MappingProxyType correctly."""
        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        data_dict = {"a": 1, "b": {"c": 2, "d": 3}}
        data_proxy = MappingProxyType(data_dict)

        normalized_dict = _normalize_monolith_for_hash(data_dict)
        normalized_proxy = _normalize_monolith_for_hash(data_proxy)

        assert normalized_dict == normalized_proxy
        assert isinstance(normalized_dict, dict)
        assert isinstance(normalized_proxy, dict)

        print("  ? Hash normalization handles MappingProxyType")

    def test_hash_normalization_deeply_nested(self):
        """Test hash normalization handles deeply nested structures."""
        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        deep_structure = {
            "level1": {
                "level2": {
                    "level3": {
                        "level4": {
                            "data": [1, 2, 3]
                        }
                    }
                }
            }
        }

        normalized = _normalize_monolith_for_hash(deep_structure)
        assert normalized["level1"]["level2"]["level3"]["level4"]["data"] == [1, 2, 3]

        print("  ? Hash normalization handles deeply nested structures")

    def test_hash_normalization_rejects_non_serializable(self):
        """Test hash normalization rejects non-JSON-serializable types."""
        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        class CustomClass:
            pass

        invalid_monolith = {
            "valid": 1,
            "invalid": CustomClass()
        }

        with pytest.raises(RuntimeError, match="non-serializable"):
            _normalize_monolith_for_hash(invalid_monolith)

        print("  ? Hash normalization rejects non-serializable types")

    def test_config_dict_includes_runtime_mode(self):
        """Test configuration dict includes runtime mode information."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        assert config.mode == RuntimeMode.PROD

        config_dict = {
            "_runtime_mode": config.mode.value,
            "_strict_mode": config.is_strict_mode(),
              "_verification_status": "verified" if config.mode == RuntimeMode.PROD else
"development",
        }

        assert config_dict["_runtime_mode"] == "prod"
        assert config_dict["_strict_mode"] is True
        assert config_dict["_verification_status"] == "verified"
        print("  ? Config dict includes runtime mode metadata")

    def test_dev_mode_allows_partial_results(self):
        """Test DEV mode sets allow_partial_results flag."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        config = RuntimeConfig.from_env()

        allow_partial = config.mode != RuntimeMode.PROD

        assert allow_partial is True
        print("  ? DEV mode allows partial results")

    def test_prod_mode_disallows_partial_results(self):
        """Test PROD mode disallows partial results."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()

        allow_partial = config.mode != RuntimeMode.PROD

        assert allow_partial is False
        print("  ? PROD mode disallows partial results")


class TestHashComputationIntegration:
    """Integration tests for hash computation in orchestrator context."""

    def test_identical_monoliths_produce_identical_hashes(self):
              """Test  identical  monoliths  produce  identical  hashes  across  multiple
computations."""
        import hashlib
        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        monolith = {
            "blocks": {
                "micro_questions": [{"id": "Q001", "text": "Test"}],
                "meso_questions": [],
                "macro_question": {},
            }
        }

        hashes = []
        for _ in range(5):
            normalized = _normalize_monolith_for_hash(monolith)
            json_str = json.dumps(
                normalized,
                sort_keys=True,
                ensure_ascii=False,
                separators=(",", ":")
            )
            hash_val = hashlib.sha256(json_str.encode("utf-8")).hexdigest()
            hashes.append(hash_val)

        assert len(set(hashes)) == 1
        print(f"  ? 5 computations produced identical hash: {hashes[0][:16]}...")

    def test_reordered_monolith_produces_same_hash(self):
        """Test reordered monolith produces same hash."""
        import hashlib
        _normalize_monolith_for_hash = _normalize_monolith_for_hash_standalone

        monolith1 = {
            "blocks": {"micro_questions": [], "meso_questions": []},
            "version": "1.0.0",
        }

        monolith2 = {
            "version": "1.0.0",
            "blocks": {"meso_questions": [], "micro_questions": []},
        }

        normalized1 = _normalize_monolith_for_hash(monolith1)
        normalized2 = _normalize_monolith_for_hash(monolith2)

                 json1  =  json.dumps(normalized1,  sort_keys=True,  ensure_ascii=False,
separators=(",", ":"))
                 json2  =  json.dumps(normalized2,  sort_keys=True,  ensure_ascii=False,
separators=(",", ":"))

        hash1 = hashlib.sha256(json1.encode("utf-8")).hexdigest()
        hash2 = hashlib.sha256(json2.encode("utf-8")).hexdigest()

        assert hash1 == hash2
        print(f"  ? Reordered monoliths produce same hash: {hash1[:16]}...")


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

tests/test_orchestrator_phase0_integration.py

```python
"""
Unit Tests for Phase 0 and Orchestrator Integration
====================================================

Tests the wiring between Phase 0 components (RuntimeConfig, exit gates)
and the Orchestrator to ensure proper integration and contract enforcement.

Test Coverage:
1. Orchestrator accepts RuntimeConfig parameter
2. Orchestrator validates RuntimeConfig in __init__
3. Orchestrator fails if Phase 0 gates failed
4. Orchestrator logs runtime mode
5. _load_configuration includes runtime mode in config
6. Phase0ValidationResult dataclass functionality
"""

import pytest
import sys
from datetime import datetime
from pathlib import Path
from unittest.mock import Mock, MagicMock, patch
from dataclasses import asdict

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

# Phase 0 imports
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from canonic_phases.Phase_zero.exit_gates import GateResult

# Orchestrator imports
from orchestration.orchestrator import Orchestrator, Phase0ValidationResult


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

@pytest.fixture
def mock_runtime_config_prod():
    """RuntimeConfig in PROD mode."""
    config = RuntimeConfig.from_dict({
        "mode": "prod",
        "allow_contradiction_fallback": False,
        "allow_validator_disable": False,
        "allow_execution_estimates": False,
        "allow_networkx_fallback": False,
        "allow_spacy_fallback": False,
        "allow_dev_ingestion_fallbacks": False,
        "allow_aggregation_defaults": False,
    })
    return config
```

```
@pytest.fixture
def mock_runtime_config_dev():
    """RuntimeConfig in DEV mode."""
    config = RuntimeConfig.from_dict({
        "mode": "dev",
        "allow_contradiction_fallback": True,
        "allow_validator_disable": True,
        "allow_execution_estimates": True,
        "allow_networkx_fallback": True,
        "allow_spacy_fallback": True,
        "allow_dev_ingestion_fallbacks": True,
        "allow_aggregation_defaults": True,
    })
    return config


@pytest.fixture
def mock_phase0_validation_success():
    """Phase0ValidationResult with all gates passed."""
    gate_results = [
        GateResult(passed=True, gate_name="bootstrap", gate_id=1),
        GateResult(passed=True, gate_name="input_verification", gate_id=2),
        GateResult(passed=True, gate_name="boot_checks", gate_id=3),
        GateResult(passed=True, gate_name="determinism", gate_id=4),
    ]
    return Phase0ValidationResult(
        all_passed=True,
        gate_results=gate_results,
        validation_time=datetime.utcnow().isoformat()
    )


@pytest.fixture
def mock_phase0_validation_failure():
    """Phase0ValidationResult with bootstrap gate failed."""
    gate_results = [
        GateResult(
            passed=False,
            gate_name="bootstrap",
            gate_id=1,
            reason="Runtime config not loaded"
        ),
    ]
    return Phase0ValidationResult(
        all_passed=False,
        gate_results=gate_results,
        validation_time=datetime.utcnow().isoformat()
    )


@pytest.fixture
def mock_orchestrator_dependencies():
    """Mock dependencies for Orchestrator initialization."""
    method_executor = Mock()
    questionnaire = Mock()
    questionnaire.data = {
        "blocks": {
            "micro_questions": [],
            "meso_questions": [],
            "macro_question": {}
        }
    }
    executor_config = Mock()

    return {
        "method_executor": method_executor,
        "questionnaire": questionnaire,
        "executor_config": executor_config,
    }


# ============================================================================
# TEST SUITE 1: Phase0ValidationResult Dataclass
# ============================================================================

class TestPhase0ValidationResult:
    """Test Phase0ValidationResult dataclass functionality."""

    def test_phase0_validation_result_all_passed(self, mock_phase0_validation_success):
        """Test Phase0ValidationResult when all gates pass."""
        result = mock_phase0_validation_success

        assert result.all_passed is True
        assert len(result.gate_results) == 4
        assert all(g.passed for g in result.gate_results)
        assert len(result.get_failed_gates()) == 0

    def test_phase0_validation_result_failure(self, mock_phase0_validation_failure):
        """Test Phase0ValidationResult when gates fail."""
        result = mock_phase0_validation_failure

        assert result.all_passed is False
        assert len(result.gate_results) == 1
        assert result.gate_results[0].passed is False
        assert len(result.get_failed_gates()) == 1
        assert result.get_failed_gates()[0].gate_name == "bootstrap"

                                def        test_phase0_validation_get_summary_success(self,
mock_phase0_validation_success):
        """Test get_summary() method for successful validation."""
        result = mock_phase0_validation_success
        summary = result.get_summary()

        assert "4/4 gates passed" in summary
        assert "failed" not in summary
```

```
                              def        test_phase0_validation_get_summary_failure(self,
mock_phase0_validation_failure):
        """Test get_summary() method for failed validation."""
        result = mock_phase0_validation_failure
        summary = result.get_summary()

        assert "0/1 gates passed" in summary or "1 gates passed" in summary
        assert "bootstrap failed" in summary


# ============================================================================
# TEST SUITE 2: Orchestrator RuntimeConfig Integration
# ============================================================================

class TestOrchestratorRuntimeConfig:
    """Test Orchestrator integration with RuntimeConfig."""

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_accepts_runtime_config(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_runtime_config_prod
    ):
        """Test that Orchestrator accepts runtime_config parameter."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=mock_runtime_config_prod
        )

        assert orchestrator.runtime_config is not None
        assert orchestrator.runtime_config.mode == RuntimeMode.PROD

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_runtime_config_none(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies
    ):
        """Test that Orchestrator works with runtime_config=None (legacy mode)."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=None
        )

        assert orchestrator.runtime_config is None

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    @patch('orchestration.orchestrator.logger')
```

```
def test_orchestrator_logs_runtime_mode_prod(
    self,
    mock_logger,
    mock_validate_structure,
    mock_validate_phases,
    mock_orchestrator_dependencies,
    mock_runtime_config_prod
):
    """Test that Orchestrator logs runtime mode (PROD)."""
    orchestrator = Orchestrator(
        **mock_orchestrator_dependencies,
        runtime_config=mock_runtime_config_prod
    )

    # Check that info was logged
    assert mock_logger.info.called
    found = False
    for call in mock_logger.info.call_args_list:
         if call.args and "orchestrator_runtime_mode" in str(call.args[0]):
             found = True
             break
         # Structlog style
         if call.args and call.args[0] == "orchestrator_runtime_mode":
             found = True
             break
    assert found, "orchestrator_runtime_mode not logged"

@patch('orchestration.orchestrator.validate_phase_definitions')
@patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
@patch('orchestration.orchestrator.logger')
def test_orchestrator_logs_runtime_mode_dev(
    self,
    mock_logger,
    mock_validate_structure,
    mock_validate_phases,
    mock_orchestrator_dependencies,
    mock_runtime_config_dev
):
    """Test that Orchestrator logs runtime mode (DEV)."""
    orchestrator = Orchestrator(
        **mock_orchestrator_dependencies,
        runtime_config=mock_runtime_config_dev
    )

    assert mock_logger.info.called
    assert orchestrator.runtime_config.mode == RuntimeMode.DEV

@patch('orchestration.orchestrator.validate_phase_definitions')
@patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
@patch('orchestration.orchestrator.logger')
def test_orchestrator_warns_if_no_runtime_config(
    self,
    mock_logger,
    mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies
    ):
        """Test that Orchestrator warns if runtime_config not provided."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=None
        )

        # Check that warning was logged
        assert mock_logger.warning.called
        found = False
        for call in mock_logger.warning.call_args_list:
             if call.args and "orchestrator_no_runtime_config" in str(call.args[0]):
                 found = True
                 break
             if call.args and call.args[0] == "orchestrator_no_runtime_config":
                 found = True
                 break
        assert found, "orchestrator_no_runtime_config warning not logged"


# ============================================================================
# TEST SUITE 3: Orchestrator Phase 0 Validation Integration
# ============================================================================

class TestOrchestratorPhase0Validation:
    """Test Orchestrator integration with Phase 0 exit gate validation."""

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_accepts_phase0_validation(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_phase0_validation_success
    ):
        """Test that Orchestrator accepts phase0_validation parameter."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            phase0_validation=mock_phase0_validation_success
        )

        assert orchestrator.phase0_validation is not None
        assert orchestrator.phase0_validation.all_passed is True

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_fails_if_phase0_gates_failed(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_phase0_validation_failure
    ):
        """Test that Orchestrator initialization fails if Phase 0 gates failed."""
        with pytest.raises(RuntimeError) as exc_info:
            Orchestrator(
                **mock_orchestrator_dependencies,
                phase0_validation=mock_phase0_validation_failure
            )

        assert "Phase 0 exit gates failed" in str(exc_info.value)
        assert "bootstrap" in str(exc_info.value)

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    @patch('orchestration.orchestrator.logger')
    def test_orchestrator_logs_phase0_validation_success(
        self,
        mock_logger,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_phase0_validation_success
    ):
        """Test that Orchestrator logs Phase 0 validation success."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            phase0_validation=mock_phase0_validation_success
        )

        # Check that info was logged
        assert mock_logger.info.called
        found = False
        for call in mock_logger.info.call_args_list:
                          if  call.args  and  "orchestrator_phase0_validation_passed"  in
str(call.args[0]):
                 found = True
                 break
             if call.args and call.args[0] == "orchestrator_phase0_validation_passed":
                 found = True
                 break
        assert found, "orchestrator_phase0_validation_passed not logged"


# ============================================================================
# TEST SUITE 4: Orchestrator _load_configuration Integration
# ============================================================================

class TestOrchestratorLoadConfiguration:
    """Test _load_configuration method with Phase 0 integration."""

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_load_configuration_includes_runtime_mode(
        self,
```

```python
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_runtime_config_prod
    ):
        """Test that _load_configuration includes runtime mode in config dict."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=mock_runtime_config_prod
        )
        orchestrator._phase_instrumentation[0] = Mock()

        config = orchestrator._load_configuration()

        assert "_runtime_mode" in config
        assert config["_runtime_mode"] == "prod"
        assert "_strict_mode" in config
        assert config["_strict_mode"] is True

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_load_configuration_without_runtime_config(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies
    ):
        """Test that _load_configuration works without runtime_config."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=None
        )
        orchestrator._phase_instrumentation[0] = Mock()

        config = orchestrator._load_configuration()

        # Should not have runtime mode keys
        assert "_runtime_mode" not in config
        assert "_strict_mode" not in config

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_load_configuration_validates_phase0_success(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_phase0_validation_success
    ):
        """Test that _load_configuration validates Phase 0 completion (success case)."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            phase0_validation=mock_phase0_validation_success
        )
        orchestrator._phase_instrumentation[0] = Mock()

        # Should not raise
        config = orchestrator._load_configuration()
        assert config is not None

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_load_configuration_fails_if_phase0_failed(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_phase0_validation_failure
    ):
        """Test that _load_configuration would fail if Phase 0 failed.

        Note: This test shows that __init__ catches Phase 0 failures,
        preventing _load_configuration from ever being called.
        """
        # __init__ should raise RuntimeError before we even get to _load_configuration
        with pytest.raises(RuntimeError) as exc_info:
            Orchestrator(
                **mock_orchestrator_dependencies,
                phase0_validation=mock_phase0_validation_failure
            )

        assert "Phase 0 exit gates failed" in str(exc_info.value)


# ============================================================================
# TEST SUITE 5: Combined Integration Tests
# ============================================================================

class TestOrchestratorFullIntegration:
    """Test full integration of RuntimeConfig + Phase 0 validation."""

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_with_full_phase0_context(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies,
        mock_runtime_config_prod,
        mock_phase0_validation_success
    ):
        """Test Orchestrator with both RuntimeConfig and Phase 0 validation."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies,
            runtime_config=mock_runtime_config_prod,
            phase0_validation=mock_phase0_validation_success
        )
        orchestrator._phase_instrumentation[0] = Mock()
        # Verify both are stored
        assert orchestrator.runtime_config is not None
        assert orchestrator.runtime_config.mode == RuntimeMode.PROD
        assert orchestrator.phase0_validation is not None
        assert orchestrator.phase0_validation.all_passed is True

        # Verify _load_configuration includes runtime mode
        config = orchestrator._load_configuration()
        assert config["_runtime_mode"] == "prod"

    @patch('orchestration.orchestrator.validate_phase_definitions')
    @patch('orchestration.questionnaire_validation._validate_questionnaire_structure')
    def test_orchestrator_backward_compatible_no_phase0(
        self,
        mock_validate_structure,
        mock_validate_phases,
        mock_orchestrator_dependencies
    ):
        """Test that Orchestrator works without any Phase 0 parameters (legacy mode)."""
        orchestrator = Orchestrator(
            **mock_orchestrator_dependencies
            # No runtime_config, no phase0_validation
        )
        orchestrator._phase_instrumentation[0] = Mock()

        assert orchestrator.runtime_config is None
        assert orchestrator.phase0_validation is None

        # Should still work
        config = orchestrator._load_configuration()
        assert config is not None
        assert "_runtime_mode" not in config


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

```
tests/test_orchestrator_signal_validation.py

"""Tests for orchestrator signal registry validation integration.

This module tests the integration of signal registry health checks into
the orchestrator initialization:
- Health check invocation in Orchestrator.__init__
- Production mode fail-fast enforcement
- Development mode warning-only behavior
- Validation result logging

Author: F.A.R.F.A.N Pipeline Team
Status: Production Test Suite
Test Priority: P1 (Critical)
"""

from typing import Any
from unittest.mock import Mock, MagicMock, patch
import pytest

from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from canonic_phases.Phase_two.executor_config import ExecutorConfig


class TestOrchestratorSignalValidationIntegration:
    """Test orchestrator integration with signal validation."""

    def test_orchestrator_calls_signal_validation_on_init(self):
        """Orchestrator calls validate_signals_for_questionnaire during init."""
        from orchestration.orchestrator import Orchestrator, MethodExecutor
        from orchestration.factory import CanonicalQuestionnaire

        # Create mock components
        mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
        mock_questionnaire.version = "1.0.0"
        mock_questionnaire.sha256 = "a" * 64
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [
                    {
                        "question_id": f"Q{i:03d}",
                        "patterns": [{"pattern": "test"}],
                        "expected_elements": [{"element": "test"}],
                        "scoring_modality": "binary_presence",
                        "validation_rules": [{"rule": "test"}],
                    }
                    for i in range(1, 301)  # 300 questions
                ]
            }
        }

        mock_signal_registry = Mock()
        mock_signal_registry.validate_signals_for_questionnaire = Mock(
            return_value={
                "valid": True,
                "total_questions": 300,
                "expected_questions": 300,
                "missing_questions": [],
                "malformed_signals": {},
                "signal_coverage": {
                    "micro_answering": {"success": 300, "failed": 0},
                    "validation": {"success": 300, "failed": 0},
                    "scoring": {"success": 300, "failed": 0},
                },
                "coverage_percentages": {
                    "micro_answering": 100.0,
                    "validation": 100.0,
                    "scoring": 100.0,
                },
                "stale_signals": [],
                "timestamp": 0.0,
                "elapsed_seconds": 1.0,
            }
        )

        mock_executor = Mock(spec=MethodExecutor)
        mock_executor.signal_registry = mock_signal_registry
        mock_executor.instances = {"test": Mock()}
        mock_executor.degraded_mode = False

        mock_runtime_config = Mock(spec=RuntimeConfig)
        mock_runtime_config.mode = RuntimeMode.DEV

        executor_config = Mock(spec=ExecutorConfig)

        # Initialize orchestrator (should call validation)
        with patch("orchestration.orchestrator._validate_questionnaire_structure"):
            orchestrator = Orchestrator(
                method_executor=mock_executor,
                questionnaire=mock_questionnaire,
                executor_config=executor_config,
                runtime_config=mock_runtime_config,
            )

        # Verify validation was called
        mock_signal_registry.validate_signals_for_questionnaire.assert_called_once()
                                                                          call_kwargs         =
mock_signal_registry.validate_signals_for_questionnaire.call_args[1]
        assert "expected_question_count" in call_kwargs

    def test_orchestrator_fails_in_prod_mode_with_invalid_signals(self):
        """Orchestrator raises RuntimeError in PROD mode when signals invalid."""
        from orchestration.orchestrator import Orchestrator, MethodExecutor
        from orchestration.factory import CanonicalQuestionnaire

        # Create mock components
        mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
        mock_questionnaire.version = "1.0.0"
        mock_questionnaire.sha256 = "a" * 64
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [
                    {"question_id": "Q001", "patterns": []},  # Invalid
                ]
            }
        }

        mock_signal_registry = Mock()
        mock_signal_registry.validate_signals_for_questionnaire = Mock(
            return_value={
                "valid": False,
                "total_questions": 1,
                "expected_questions": 300,
                "missing_questions": ["Q001"],
                "malformed_signals": {"Q001": ["missing patterns"]},
                "signal_coverage": {
                    "micro_answering": {"success": 0, "failed": 1},
                },
                "coverage_percentages": {
                    "micro_answering": 0.0,
                },
                "stale_signals": [],
                "timestamp": 0.0,
                "elapsed_seconds": 0.1,
            }
        )

        mock_executor = Mock(spec=MethodExecutor)
        mock_executor.signal_registry = mock_signal_registry
        mock_executor.instances = {"test": Mock()}

        # PROD mode
        mock_runtime_config = Mock(spec=RuntimeConfig)
        mock_runtime_config.mode = Mock(value="prod")

        executor_config = Mock(spec=ExecutorConfig)

        # Should raise RuntimeError in PROD mode
        with pytest.raises(RuntimeError) as exc_info:
            with patch("orchestration.orchestrator._validate_questionnaire_structure"):
                Orchestrator(
                    method_executor=mock_executor,
                    questionnaire=mock_questionnaire,
                    executor_config=executor_config,
                    runtime_config=mock_runtime_config,
                )

        # Check error message mentions production mode
                assert  "Production  mode"  in  str(exc_info.value)  or  "production"  in
str(exc_info.value).lower()

    def test_orchestrator_warns_in_dev_mode_with_invalid_signals(self, caplog):
```

```
"""Orchestrator logs warning in DEV mode but continues."""
from orchestration.orchestrator import Orchestrator, MethodExecutor
from orchestration.factory import CanonicalQuestionnaire

# Create mock components
mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
mock_questionnaire.version = "1.0.0"
mock_questionnaire.sha256 = "a" * 64
mock_questionnaire.data = {
    "blocks": {
        "micro_questions": [
            {
                "question_id": "Q001",
                "patterns": [{"pattern": "test"}],
                "expected_elements": [{"element": "test"}],
                "scoring_modality": "binary_presence",
                "validation_rules": [{"rule": "test"}],
            }
        ]
    }
}

mock_signal_registry = Mock()
mock_signal_registry.validate_signals_for_questionnaire = Mock(
    return_value={
        "valid": False,
        "total_questions": 1,
        "expected_questions": 300,
        "missing_questions": [],
        "malformed_signals": {},
        "signal_coverage": {
            "micro_answering": {"success": 1, "failed": 0},
        },
        "coverage_percentages": {
            "micro_answering": 100.0,
        },
        "stale_signals": [],
        "timestamp": 0.0,
        "elapsed_seconds": 0.1,
    }
)

mock_executor = Mock(spec=MethodExecutor)
mock_executor.signal_registry = mock_signal_registry
mock_executor.instances = {"test": Mock()}

# DEV mode
mock_runtime_config = Mock(spec=RuntimeConfig)
mock_runtime_config.mode = RuntimeMode.DEV

executor_config = Mock(spec=ExecutorConfig)

# Should succeed but log warning
with patch("orchestration.orchestrator._validate_questionnaire_structure"):
```

```
            orchestrator = Orchestrator(
                method_executor=mock_executor,
                questionnaire=mock_questionnaire,
                executor_config=executor_config,
                runtime_config=mock_runtime_config,
            )

        # Orchestrator should be created successfully
        assert orchestrator is not None

    def test_orchestrator_logs_validation_success(self, caplog):
        """Orchestrator logs when signal validation passes."""
        from orchestration.orchestrator import Orchestrator, MethodExecutor
        from orchestration.factory import CanonicalQuestionnaire

        # Create mock components with valid signals
        mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
        mock_questionnaire.version = "1.0.0"
        mock_questionnaire.sha256 = "a" * 64
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [
                    {
                        "question_id": f"Q{i:03d}",
                        "patterns": [{"pattern": "test"}],
                        "expected_elements": [{"element": "test"}],
                        "scoring_modality": "binary_presence",
                        "validation_rules": [{"rule": "test"}],
                    }
                    for i in range(1, 301)
                ]
            }
        }

        mock_signal_registry = Mock()
        mock_signal_registry.validate_signals_for_questionnaire = Mock(
            return_value={
                "valid": True,
                "total_questions": 300,
                "expected_questions": 300,
                "missing_questions": [],
                "malformed_signals": {},
                "signal_coverage": {
                    "micro_answering": {"success": 300, "failed": 0},
                    "validation": {"success": 300, "failed": 0},
                    "scoring": {"success": 300, "failed": 0},
                },
                "coverage_percentages": {
                    "micro_answering": 100.0,
                    "validation": 100.0,
                    "scoring": 100.0,
                },
                "stale_signals": [],
                "timestamp": 0.0,
                "elapsed_seconds": 2.5,
            }
        )

        mock_executor = Mock(spec=MethodExecutor)
        mock_executor.signal_registry = mock_signal_registry
        mock_executor.instances = {"test": Mock()}

        mock_runtime_config = Mock(spec=RuntimeConfig)
        mock_runtime_config.mode = RuntimeMode.PROD

        executor_config = Mock(spec=ExecutorConfig)

        # Initialize orchestrator
        with patch("orchestration.orchestrator._validate_questionnaire_structure"):
            orchestrator = Orchestrator(
                method_executor=mock_executor,
                questionnaire=mock_questionnaire,
                executor_config=executor_config,
                runtime_config=mock_runtime_config,
            )

        # Orchestrator should be created successfully
        assert orchestrator is not None


@pytest.mark.updated
class TestOrchestratorPhase3SignalTracking:
    """Test Phase 3 signal tracking in scoring details."""

    def test_phase3_records_applied_signals_in_scoring_details(self):
        """Phase 3 scoring records which signals were applied."""
        # This is an integration point test - verify that scoring_details
        # contains signal tracking information

        from orchestration.orchestrator import ScoredMicroQuestion

        # Create a mock scored result as it would come from Phase 3
        scoring_details = {
            "source": "evidence_nexus",
            "method": "overall_confidence",
            "signal_enrichment_raw": {
                "modality": "binary_presence",
                "source_hash": "abc123",
                "signal_source": "sisas_registry",
            },
            "applied_signals": {
                "question_id": "Q001",
                "scoring_modality": "binary_presence",
                "has_modality_config": True,
                "threshold_defined": True,
                "signal_lookup_timestamp": 1234567890.0,
            },
        }
        # Verify structure exists
        assert "applied_signals" in scoring_details
        assert "question_id" in scoring_details["applied_signals"]
        assert "scoring_modality" in scoring_details["applied_signals"]
        assert "signal_lookup_timestamp" in scoring_details["applied_signals"]

    def test_phase3_logs_signal_application_debug(self, caplog):
        """Phase 3 logs signal application at debug level."""
        # This test verifies that the logging structure is correct
        # In actual integration, we'd see these logs during Phase 3

        import structlog
        logger = structlog.get_logger(__name__)

        with caplog.at_level("DEBUG"):
            logger.debug(
                "signal_applied_in_scoring",
                question_id="Q001",
                modality="binary_presence",
                scoring_modality="binary_presence",
            )

        # Check that the log was created
        assert len(caplog.records) > 0


@pytest.mark.updated
class TestOrchestratorSignalValidationRobustness:
    """Test robustness of signal validation in orchestrator."""

    def test_orchestrator_handles_validation_exception_gracefully(self):
        """Orchestrator handles exceptions during validation gracefully."""
        from orchestration.orchestrator import Orchestrator, MethodExecutor
        from orchestration.factory import CanonicalQuestionnaire

        # Create mock components
        mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
        mock_questionnaire.version = "1.0.0"
        mock_questionnaire.sha256 = "a" * 64
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [
                    {"question_id": "Q001", "patterns": [{"pattern": "test"}]},
                ]
            }
        }

        mock_signal_registry = Mock()
        # Validation method raises exception
        mock_signal_registry.validate_signals_for_questionnaire = Mock(
            side_effect=Exception("Validation internal error")
        )
        mock_executor = Mock(spec=MethodExecutor)
        mock_executor.signal_registry = mock_signal_registry
        mock_executor.instances = {"test": Mock()}

        mock_runtime_config = Mock(spec=RuntimeConfig)
        mock_runtime_config.mode = RuntimeMode.DEV

        executor_config = Mock(spec=ExecutorConfig)

        # Should raise the exception (not silently catch it)
        with pytest.raises(Exception) as exc_info:
            with patch("orchestration.orchestrator._validate_questionnaire_structure"):
                Orchestrator(
                    method_executor=mock_executor,
                    questionnaire=mock_questionnaire,
                    executor_config=executor_config,
                    runtime_config=mock_runtime_config,
                )

        assert "Validation internal error" in str(exc_info.value)

    def test_orchestrator_requires_signal_registry(self):
        """Orchestrator fails if signal_registry is None."""
        from orchestration.orchestrator import Orchestrator, MethodExecutor
        from orchestration.factory import CanonicalQuestionnaire

        mock_questionnaire = Mock(spec=CanonicalQuestionnaire)
        mock_questionnaire.version = "1.0.0"
        mock_questionnaire.sha256 = "a" * 64
        mock_questionnaire.data = {"blocks": {"micro_questions": []}}

        mock_executor = Mock(spec=MethodExecutor)
        mock_executor.signal_registry = None  # No signal registry
        mock_executor.instances = {"test": Mock()}

        executor_config = Mock(spec=ExecutorConfig)

        # Should raise RuntimeError about missing signal_registry
        with pytest.raises(RuntimeError) as exc_info:
            with patch("orchestration.orchestrator._validate_questionnaire_structure"):
                Orchestrator(
                    method_executor=mock_executor,
                    questionnaire=mock_questionnaire,
                    executor_config=executor_config,
                )

        assert "signal_registry" in str(exc_info.value).lower()
```

tests/test_phase0_complete.py

```
"""
Comprehensive Test Suite for Phase 0: Input Validation
========================================================

Tests all components identified in the Phase 0 analysis:
1. Bootstrap integrity
2. Input verification (PDF + Questionnaire)
3. Boot checks (dependencies)
4. Runtime configuration
5. Contract validation
6. Hash computation
7. Verification manifest generation
8. Claims logging

Author: F.A.R.F.A.N Test Suite
Date: 2025-12-10
"""

import hashlib
import json
import os
import tempfile
from datetime import datetime, timezone
from pathlib import Path
from unittest.mock import Mock, patch

import pytest

# Phase 0 components
from canonic_phases.Phase_one.phase0_input_validation import (
    CanonicalInput,
    CanonicalInputValidator,
    Phase0Input,
    Phase0InputValidator,
    Phase0ValidationContract,
    PHASE0_VERSION,
)
from canonic_phases.Phase_zero.boot_checks import (
    BootCheckError,
    check_calibration_files,
    check_contradiction_module_available,
    check_networkx_available,
    check_spacy_model_available,
    check_wiring_validator_available,
    get_boot_check_summary,
    run_boot_checks,
)
from canonic_phases.Phase_zero.runtime_config import (
    ConfigurationError,
    FallbackCategory,
    RuntimeConfig,
    RuntimeMode,
    get_runtime_config,
    reset_runtime_config,
)


# ==================================================================================
# TEST SUITE 1: RUNTIME CONFIGURATION
# ==================================================================================


class TestRuntimeConfiguration:
    """Test suite for RuntimeConfig validation and parsing."""

    def setup_method(self):
        """Reset config before each test."""
        reset_runtime_config()
        # Clear environment
        for key in list(os.environ.keys()):
            if key.startswith("SAAAAAA_") or key.startswith("ALLOW_"):
                del os.environ[key]

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

    def test_default_prod_mode(self):
        """Test default runtime mode is PROD."""
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.PROD
        print("  ? Default runtime mode is PROD")

    def test_dev_mode_parsing(self):
        """Test DEV mode parsing from environment."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.DEV
        print("  ? DEV mode parsed correctly")

    def test_exploratory_mode_parsing(self):
        """Test EXPLORATORY mode parsing."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "exploratory"
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.EXPLORATORY
        print("  ? EXPLORATORY mode parsed correctly")

    def test_invalid_mode_raises_error(self):
        """Test invalid runtime mode raises ConfigurationError."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "invalid_mode"
        with pytest.raises(ConfigurationError, match="Invalid SAAAAAA_RUNTIME_MODE"):
            RuntimeConfig.from_env()
        print("  ? Invalid mode raises ConfigurationError")

    def test_prod_illegal_combination_dev_ingestion(self):
        """Test PROD + ALLOW_DEV_INGESTION_FALLBACKS raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + DEV_INGESTION_FALLBACKS rejected")

    def test_prod_illegal_combination_execution_estimates(self):
        """Test PROD + ALLOW_EXECUTION_ESTIMATES raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_EXECUTION_ESTIMATES"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + EXECUTION_ESTIMATES rejected")

    def test_prod_illegal_combination_aggregation_defaults(self):
        """Test PROD + ALLOW_AGGREGATION_DEFAULTS raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + AGGREGATION_DEFAULTS rejected")

    def test_dev_allows_all_fallbacks(self):
        """Test DEV mode allows all fallback flags."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"
        os.environ["ALLOW_EXECUTION_ESTIMATES"] = "true"
        os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"

        config = RuntimeConfig.from_env()
        assert config.allow_dev_ingestion_fallbacks
        assert config.allow_execution_estimates
        assert config.allow_aggregation_defaults
        print("  ? DEV mode allows all fallbacks")

    def test_strict_mode_detection(self):
        """Test is_strict_mode() correctly identifies strict PROD."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()
        assert config.is_strict_mode()
        print("  ? Strict mode detected correctly")

    def test_fallback_summary_generation(self):
        """Test get_fallback_summary() returns correct structure."""
        config = RuntimeConfig.from_env()
        summary = config.get_fallback_summary()

        assert "critical" in summary
        assert "quality" in summary
        assert "development" in summary
        assert "operational" in summary
        print("  ? Fallback summary generated correctly")

    def test_timeout_parsing(self):
        """Test phase timeout parsing from environment."""
        os.environ["PHASE_TIMEOUT_SECONDS"] = "600"
        config = RuntimeConfig.from_env()
        assert config.phase_timeout_seconds == 600
        print("  ? Phase timeout parsed correctly")

    def test_expected_counts_parsing(self):
        """Test expected question/method counts."""
        os.environ["EXPECTED_QUESTION_COUNT"] = "305"
        os.environ["EXPECTED_METHOD_COUNT"] = "416"
        config = RuntimeConfig.from_env()
        assert config.expected_question_count == 305
        assert config.expected_method_count == 416
        print("  ? Expected counts parsed correctly")


# ============================================================================
# TEST SUITE 2: INPUT CONTRACT VALIDATION
# ============================================================================


class TestPhase0InputContract:
    """Test suite for Phase0Input and Pydantic validation."""

    def test_phase0_input_creation(self):
        """Test Phase0Input dataclass creation."""
        pdf_path = Path("/tmp/test.pdf")
        input_data = Phase0Input(
            pdf_path=pdf_path,
            run_id="20251210_120000",
            questionnaire_path=None,
        )

        assert input_data.pdf_path == pdf_path
        assert input_data.run_id == "20251210_120000"
        assert input_data.questionnaire_path is None
        print("  ? Phase0Input created successfully")

    def test_phase0_input_validator_strict_mode(self):
        """Test Phase0InputValidator uses StrictModel pattern."""
        config = Phase0InputValidator.model_config

        assert config.get("extra") == "forbid"
        assert config.get("validate_assignment") is True
        assert config.get("str_strip_whitespace") is True
        print("  ? Phase0InputValidator uses StrictModel pattern")

    def test_phase0_input_validator_empty_pdf_path(self):
        """Test validator rejects empty pdf_path."""
          with pytest.raises(Exception, match=r"(pdf_path cannot be empty|String should
have at least 1 character)"):
            Phase0InputValidator(pdf_path="", run_id="test_run")
        print("  ? Empty pdf_path rejected")
    def test_phase0_input_validator_empty_run_id(self):
        """Test validator rejects empty run_id."""
         with pytest.raises(Exception, match=r"(run_id cannot be empty|String should have
at least 1 character)"):
            Phase0InputValidator(pdf_path="/tmp/test.pdf", run_id="")
        print("  ? Empty run_id rejected")

    def test_phase0_input_validator_invalid_run_id_characters(self):
        """Test validator rejects run_id with invalid filesystem characters."""
        invalid_chars = ['/', '\\', ':', '*', '?', '"', '<', '>', '|']
        for char in invalid_chars:
            with pytest.raises(Exception, match="invalid characters"):
                Phase0InputValidator(
                    pdf_path="/tmp/test.pdf",
                    run_id=f"test{char}run"
                )
        print(f"  ? Invalid run_id characters rejected ({len(invalid_chars)} tested)")

    def test_phase0_input_validator_valid_input(self):
        """Test validator accepts valid input."""
        validator = Phase0InputValidator(
            pdf_path="/tmp/test.pdf",
            run_id="20251210_120000",
            questionnaire_path="/tmp/questionnaire.json",
        )

        assert validator.pdf_path == "/tmp/test.pdf"
        assert validator.run_id == "20251210_120000"
        print("  ? Valid input accepted")


class TestCanonicalInputContract:
    """Test suite for CanonicalInput output contract."""

    def test_canonical_input_creation(self):
        """Test CanonicalInput dataclass creation."""
        output = CanonicalInput(
            document_id="Plan_1",
            run_id="20251210_120000",
            pdf_path=Path("/tmp/Plan_1.pdf"),
            pdf_sha256="a" * 64,
            pdf_size_bytes=1024,
            pdf_page_count=10,
            questionnaire_path=Path("/tmp/questionnaire.json"),
            questionnaire_sha256="b" * 64,
            created_at=datetime.now(timezone.utc),
            phase0_version=PHASE0_VERSION,
            validation_passed=True,
            validation_errors=[],
            validation_warnings=[],
        )

        assert output.document_id == "Plan_1"
        assert output.validation_passed is True
        assert len(output.validation_errors) == 0
        print("  ? CanonicalInput created successfully")

    def test_canonical_input_validator_validation_passed_false(self):
        """Test validator rejects validation_passed=False."""
        with pytest.raises(Exception, match="validation_passed must be True"):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="a" * 64,
                pdf_size_bytes=1024,
                pdf_page_count=10,
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=False,
                validation_errors=[],
                validation_warnings=[],
            )
        print("  ? validation_passed=False rejected")

    def test_canonical_input_validator_errors_with_passed_true(self):
        """Test validator rejects errors when validation_passed=True."""
        with pytest.raises(Exception, match="validation_errors is not empty"):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="a" * 64,
                pdf_size_bytes=1024,
                pdf_page_count=10,
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=True,
                validation_errors=["Some error"],
                validation_warnings=[],
            )
        print("  ? Inconsistent validation state rejected")

    def test_canonical_input_validator_invalid_sha256_length(self):
        """Test validator rejects invalid SHA256 length."""
         with pytest.raises(Exception, match=r"(must be 64 characters|String should have
at least 64 characters)"):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="abc123",  # Too short
                pdf_size_bytes=1024,
                pdf_page_count=10,
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=True,
                validation_errors=[],
                validation_warnings=[],
            )
        print("  ? Invalid SHA256 length rejected")

    def test_canonical_input_validator_invalid_sha256_format(self):
        """Test validator rejects non-hexadecimal SHA256."""
        with pytest.raises(Exception, match="must be hexadecimal"):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="z" * 64,  # Invalid hex
                pdf_size_bytes=1024,
                pdf_page_count=10,
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=True,
                validation_errors=[],
                validation_warnings=[],
            )
        print("  ? Non-hexadecimal SHA256 rejected")

    def test_canonical_input_validator_zero_size(self):
        """Test validator rejects zero file size."""
        with pytest.raises(Exception):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="a" * 64,
                pdf_size_bytes=0,  # Invalid
                pdf_page_count=10,
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=True,
                validation_errors=[],
                validation_warnings=[],
            )
        print("  ? Zero file size rejected")

    def test_canonical_input_validator_zero_pages(self):
        """Test validator rejects zero page count."""
        with pytest.raises(Exception):
            CanonicalInputValidator(
                document_id="test",
                run_id="test_run",
                pdf_path="/tmp/test.pdf",
                pdf_sha256="a" * 64,
                pdf_size_bytes=1024,
                pdf_page_count=0,  # Invalid
                questionnaire_path="/tmp/q.json",
                questionnaire_sha256="b" * 64,
                created_at="2025-12-10T12:00:00Z",
                phase0_version="1.0.0",
                validation_passed=True,
                validation_errors=[],
                validation_warnings=[],
            )
        print("  ? Zero page count rejected")


# ============================================================================
# TEST SUITE 3: HASH COMPUTATION
# ============================================================================


class TestHashComputation:
    """Test suite for SHA256 hash computation."""

    def test_sha256_computation_deterministic(self):
        """Test SHA256 computation is deterministic."""
        with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.txt') as f:
            f.write("Test content for hash computation")
            temp_path = Path(f.name)

        try:
            contract = Phase0ValidationContract()
            hash1 = contract._compute_sha256(temp_path)
            hash2 = contract._compute_sha256(temp_path)

            assert hash1 == hash2
            assert len(hash1) == 64
            assert all(c in "0123456789abcdef" for c in hash1)
            print(f"  ? SHA256 computation is deterministic: {hash1[:16]}...")
        finally:
            temp_path.unlink()

    def test_sha256_different_content(self):
        """Test different content produces different hashes."""
        with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.txt') as f1:
            f1.write("Content 1")
            path1 = Path(f1.name)

        with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.txt') as f2:
            f2.write("Content 2")
            path2 = Path(f2.name)

        try:
            contract = Phase0ValidationContract()
            hash1 = contract._compute_sha256(path1)
            hash2 = contract._compute_sha256(path2)

            assert hash1 != hash2
            print(f"  ? Different content ? different hashes")
        finally:
            path1.unlink()
            path2.unlink()

    def test_sha256_matches_hashlib(self):
        """Test hash matches standard hashlib computation."""
        content = b"Test content for verification"
        expected_hash = hashlib.sha256(content).hexdigest()

        with tempfile.NamedTemporaryFile(mode='wb', delete=False) as f:
            f.write(content)
            temp_path = Path(f.name)

        try:
            contract = Phase0ValidationContract()
            computed_hash = contract._compute_sha256(temp_path)

            assert computed_hash == expected_hash
            print(f"  ? Hash matches hashlib: {expected_hash[:16]}...")
        finally:
            temp_path.unlink()


# ============================================================================
# TEST SUITE 4: BOOT CHECKS
# ============================================================================


class TestBootChecks:
    """Test suite for boot-time dependency checks."""

    def setup_method(self):
        """Setup test environment."""
        reset_runtime_config()

    def test_networkx_available(self):
        """Test NetworkX availability check."""
        result = check_networkx_available()
        # Should be True if networkx is installed
        print(f"  ? NetworkX availability: {result}")

    def test_boot_check_summary_format(self):
        """Test boot check summary formatting."""
        results = {
            "check1": True,
            "check2": False,
            "check3": True,
        }
        summary = get_boot_check_summary(results)

        assert "Boot Checks: 2/3 passed" in summary
        assert "? check1" in summary
        assert "? check2" in summary
        assert "? check3" in summary
        print("  ? Boot check summary formatted correctly")

    def test_boot_check_error_structure(self):
        """Test BootCheckError has correct structure."""
        error = BootCheckError(
            component="test_component",
            reason="Test failure reason",
            code="TEST_ERROR_CODE",
        )

        assert error.component == "test_component"
        assert error.reason == "Test failure reason"
        assert error.code == "TEST_ERROR_CODE"
        assert "TEST_ERROR_CODE" in str(error)
        print("  ? BootCheckError structure correct")

    @patch('canonic_phases.Phase_zero.boot_checks.importlib.import_module')
    def test_contradiction_module_check_prod_strict(self, mock_import):
        """Test contradiction module check fails in PROD mode."""
        mock_import.side_effect = ImportError("Module not found")

        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_CONTRADICTION_FALLBACK"] = "false"
        config = RuntimeConfig.from_env()

        with pytest.raises(BootCheckError, match="CONTRADICTION_MODULE_MISSING"):
            check_contradiction_module_available(config)
        print("  ? Contradiction module check strict in PROD")

    def test_contradiction_module_check_dev_permissive(self):
        """Test contradiction module check allows fallback in DEV."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        config = RuntimeConfig.from_env()

        # Should not raise, just return False
          with patch('canonic_phases.Phase_zero.boot_checks.importlib.import_module') as
mock_import:
            mock_import.side_effect = ImportError("Module not found")
            result = check_contradiction_module_available(config)
            assert result is False
        print("  ? Contradiction module check permissive in DEV")


# ============================================================================
# TEST SUITE 5: PHASE 0 CONTRACT EXECUTION
# ============================================================================
```

```python
class TestPhase0ContractExecution:
    """Test suite for Phase0ValidationContract.execute()."""

    @pytest.mark.asyncio
    async def test_contract_execution_pdf_not_found(self):
        """Test contract execution fails if PDF not found."""
        input_data = Phase0Input(
            pdf_path=Path("/nonexistent/file.pdf"),
            run_id="20251210_120000",
        )

        contract = Phase0ValidationContract()

        with pytest.raises(FileNotFoundError, match="PDF not found"):
            await contract.execute(input_data)
        print("  ? PDF not found error raised")

    @pytest.mark.asyncio
    async def test_contract_execution_questionnaire_not_found(self):
        """Test contract execution fails if questionnaire not found."""
        # Create temporary PDF
        with tempfile.NamedTemporaryFile(mode='wb', delete=False, suffix='.pdf') as f:
            # Write minimal PDF header
            f.write(b"%PDF-1.4\n")
            pdf_path = Path(f.name)

        try:
            input_data = Phase0Input(
                pdf_path=pdf_path,
                run_id="20251210_120000",
                questionnaire_path=Path("/nonexistent/questionnaire.json"),
            )

            contract = Phase0ValidationContract()

            with pytest.raises(FileNotFoundError, match="Questionnaire not found"):
                await contract.execute(input_data)
            print("  ? Questionnaire not found error raised")
        finally:
            pdf_path.unlink()

    def test_contract_invariants_registered(self):
        """Test Phase0ValidationContract has required invariants."""
        contract = Phase0ValidationContract()

        invariant_names = [inv.name for inv in contract.invariants]

        assert "validation_passed" in invariant_names
        assert "pdf_page_count_positive" in invariant_names
        assert "pdf_size_positive" in invariant_names
        assert "sha256_format" in invariant_names
        assert "no_validation_errors" in invariant_names
        print(f"  ? {len(invariant_names)} invariants registered")
```

```
# ============================================================================
# TEST SUITE 6: INTEGRATION TESTS
# ============================================================================


class TestPhase0Integration:
    """Integration tests for complete Phase 0 flow."""

    def test_phase0_version_constant(self):
        """Test PHASE0_VERSION constant is defined."""
        assert PHASE0_VERSION == "1.0.0"
        print(f"  ? Phase 0 version: {PHASE0_VERSION}")

    def test_fallback_categories_defined(self):
        """Test all fallback categories are defined."""
        categories = list(FallbackCategory)

        assert FallbackCategory.CRITICAL in categories
        assert FallbackCategory.QUALITY in categories
        assert FallbackCategory.DEVELOPMENT in categories
        assert FallbackCategory.OPERATIONAL in categories
        print(f"  ? {len(categories)} fallback categories defined")

    def test_runtime_modes_defined(self):
        """Test all runtime modes are defined."""
        modes = list(RuntimeMode)

        assert RuntimeMode.PROD in modes
        assert RuntimeMode.DEV in modes
        assert RuntimeMode.EXPLORATORY in modes
        print(f"  ? {len(modes)} runtime modes defined")


# ============================================================================
# TEST RUNNER
# ============================================================================


def run_all_tests():
    """Run all Phase 0 tests with detailed output."""
    print("\n" + "=" * 80)
    print("PHASE 0 COMPREHENSIVE TEST SUITE")
    print("=" * 80)

    test_classes = [
        TestRuntimeConfiguration,
        TestPhase0InputContract,
        TestCanonicalInputContract,
        TestHashComputation,
        TestBootChecks,
        TestPhase0ContractExecution,
        TestPhase0Integration,
    ]

    total_passed = 0
    total_failed = 0

    for test_class in test_classes:
        print(f"\n{test_class.__name__}:")
        print("-" * 80)

        # Get all test methods
        test_methods = [
            method for method in dir(test_class)
            if method.startswith("test_")
        ]

        for method_name in test_methods:
            try:
                instance = test_class()
                if hasattr(instance, 'setup_method'):
                    instance.setup_method()

                method = getattr(instance, method_name)
                method()

                if hasattr(instance, 'teardown_method'):
                    instance.teardown_method()

                total_passed += 1
            except Exception as e:
                print(f"  ? {method_name}: {e}")
                total_failed += 1

    print("\n" + "=" * 80)
    print(f"RESULTS: {total_passed} passed, {total_failed} failed")
    print("=" * 80)

    return total_passed, total_failed


if __name__ == "__main__":
    run_all_tests()
```

tests/test_phase0_damaged_artifacts.py

```python
"""
Integration Tests for Phase 0 with Damaged Artifacts
====================================================

Tests Phase 0 validation with intentionally corrupted or missing artifacts:
- Corrupted questionnaire files
- Missing methods in registry
- Failed smoke test methods
- Invalid hash configurations

These tests require a fully configured environment with all dependencies.
Run after: pip install -r requirements.txt

Test Coverage:
1. Corrupted questionnaire detection
2. Missing method detection
3. Failed smoke test detection
4. End-to-end orchestrator initialization failures
"""

import pytest
import sys
import os
import json
import hashlib
import tempfile
from pathlib import Path
from unittest.mock import Mock, patch

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

# Mark all tests in this module as integration tests
pytestmark = pytest.mark.integration


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

@pytest.fixture
def temp_questionnaire_file():
    """Create a temporary questionnaire file."""
    with tempfile.NamedTemporaryFile(mode='w', suffix='.json', delete=False) as f:
        questionnaire_data = {
            "blocks": {
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:03d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO"}
            }
        }
        json.dump(questionnaire_data, f)
        temp_path = Path(f.name)

    yield temp_path

    # Cleanup
    if temp_path.exists():
        temp_path.unlink()


@pytest.fixture
def corrupted_questionnaire_file():
    """Create a corrupted questionnaire file (invalid JSON)."""
    with tempfile.NamedTemporaryFile(mode='w', suffix='.json', delete=False) as f:
        f.write("{ invalid json content ][")
        temp_path = Path(f.name)

    yield temp_path

    # Cleanup
    if temp_path.exists():
        temp_path.unlink()


@pytest.fixture
def questionnaire_with_wrong_hash(temp_questionnaire_file):
    """Get questionnaire file with computed hash, but set wrong expected hash."""
    actual_hash = hashlib.sha256(temp_questionnaire_file.read_bytes()).hexdigest()
    wrong_hash = "f" * 64  # Intentionally wrong

    return temp_questionnaire_file, actual_hash, wrong_hash


# ============================================================================
# TEST SUITE 1: Corrupted Questionnaire Detection
# ============================================================================

@pytest.mark.skipif(
    "SKIP_INTEGRATION_TESTS" in os.environ,
    reason="Integration tests require full dependency installation"
)
class TestCorruptedQuestionnaireDetection:
    """Test Phase 0 detection of corrupted questionnaires."""

                        def      test_orchestrator_rejects_corrupted_questionnaire(self,
corrupted_questionnaire_file):
        """Test that orchestrator rejects corrupted questionnaire during Phase 0."""
              pytest.skip("Requires  full  orchestrator  dependencies  -  implement  after
environment setup")

        # This test would:
        # 1. Try to initialize orchestrator with corrupted questionnaire
        # 2. Expect Phase0ValidationResult to fail
        # 3. Verify error message mentions "questionnaire" corruption
        # 4. Ensure orchestrator.__init__ raises RuntimeError
```

```python
    def test_phase0_runner_detects_invalid_questionnaire_hash(
        self,
        questionnaire_with_wrong_hash
    ):
        """Test that Phase 0 runner detects questionnaire hash mismatch."""
        pytest.skip("Requires Phase 0 runner setup - implement after environment ready")

        # This test would:
        # 1. Configure EXPECTED_QUESTIONNAIRE_SHA256 to wrong hash
        # 2. Run Phase 0 validation
        # 3. Expect gate 5 (questionnaire_integrity) to fail
        # 4. Verify fail-fast stops at gate 5

    def test_dev_mode_logs_but_continues_with_hash_mismatch(
        self,
        questionnaire_with_wrong_hash
    ):
        """Test that DEV mode logs questionnaire mismatch but doesn't block."""
        pytest.skip("Requires full environment - implement when dependencies available")

        # This test would:
        # 1. Set RuntimeMode to DEV
        # 2. Configure wrong expected hash
        # 3. Run Phase 0
        # 4. Verify gate passes with warning
        # 5. Check that orchestrator can still initialize (degraded mode)


# ============================================================================
# TEST SUITE 2: Missing Methods Detection
# ============================================================================

@pytest.mark.skipif(
    "SKIP_INTEGRATION_TESTS" in os.environ,
    reason="Integration tests require full dependency installation"
)
class TestMissingMethodsDetection:
    """Test Phase 0 detection of missing methods."""

    def test_orchestrator_rejects_insufficient_method_count(self):
        """Test that orchestrator rejects when method count below threshold."""
        pytest.skip("Requires method registry mocking - implement after setup")

        # This test would:
        # 1. Mock MethodRegistry.get_stats() to return count < 416
        # 2. Try to initialize orchestrator
        # 3. Expect Phase0ValidationResult gate 6 to fail
        # 4. Verify orchestrator.__init__ raises RuntimeError in PROD mode

    def test_prod_mode_rejects_failed_method_classes(self):
        """Test that PROD mode rejects any failed method classes."""
        pytest.skip("Requires full orchestrator setup")
        # This test would:
        # 1. Mock MethodRegistry with some failed classes
        # 2. Set RuntimeMode to PROD
        # 3. Run Phase 0 validation
        # 4. Expect gate 6 to fail with failed_classes reason

    def test_dev_mode_allows_degraded_method_registry(self):
        """Test that DEV mode allows degraded method registry with warnings."""
        pytest.skip("Requires full orchestrator setup")

        # This test would:
        # 1. Mock MethodRegistry with some failed classes
        # 2. Set RuntimeMode to DEV
        # 3. Run Phase 0 validation
        # 4. Verify gate 6 passes with warning
        # 5. Check orchestrator initializes in degraded mode


# ============================================================================
# TEST SUITE 3: Failed Smoke Tests Detection
# ============================================================================

@pytest.mark.skipif(
    "SKIP_INTEGRATION_TESTS" in os.environ,
    reason="Integration tests require full dependency installation"
)
class TestFailedSmokeTestsDetection:
    """Test Phase 0 smoke test failures."""

    def test_orchestrator_rejects_missing_ingest_methods(self):
        """Test that orchestrator rejects when ingest category methods missing."""
        pytest.skip("Requires method executor mocking")

        # This test would:
        # 1. Mock MethodExecutor with missing PDFChunkExtractor
        # 2. Run Phase 0 validation
        # 3. Expect gate 7 to fail with "ingest" in reason

    def test_orchestrator_rejects_missing_scoring_methods(self):
        """Test that orchestrator rejects when scoring category methods missing."""
        pytest.skip("Requires method executor mocking")

        # This test would:
        # 1. Mock MethodExecutor with missing SemanticAnalyzer
        # 2. Run Phase 0 validation
        # 3. Expect gate 7 to fail with "scoring" in reason

    def test_orchestrator_rejects_missing_aggregation_methods(self):
        """Test that orchestrator rejects when aggregation methods missing."""
        pytest.skip("Requires method executor mocking")

        # This test would:
        # 1. Mock MethodExecutor with missing DimensionAggregator
        # 2. Run Phase 0 validation
        # 3. Expect gate 7 to fail with "aggregation" in reason

    def test_dev_mode_allows_failed_smoke_tests(self):
        """Test that DEV mode allows failed smoke tests with warnings."""
        pytest.skip("Requires full setup")

        # This test would:
        # 1. Mock MethodExecutor with missing smoke test methods
        # 2. Set RuntimeMode to DEV
        # 3. Run Phase 0 validation
        # 4. Verify gate 7 passes with warning
        # 5. Check orchestrator initializes with degraded capabilities


# ==============================================================================
# TEST SUITE 4: End-to-End Orchestrator Initialization
# ==============================================================================

@pytest.mark.skipif(
    "SKIP_INTEGRATION_TESTS" in os.environ,
    reason="Integration tests require full dependency installation"
)
class TestOrchestratorInitializationWithDamagedArtifacts:
    """Test end-to-end orchestrator initialization with damaged artifacts."""

    def test_orchestrator_refuses_construction_without_phase0_validation(self):
                         """Test   that   orchestrator   refuses   construction   without
Phase0ValidationResult."""
        pytest.skip("Requires full orchestrator dependencies")

        # This test would:
        # 1. Try to construct Orchestrator with phase0_validation=None
        # 2. In PROD mode, expect initialization to proceed (legacy compatibility)
        # 3. Verify warning is logged about missing validation

    def test_orchestrator_refuses_construction_with_failed_phase0(self):
        """Test that orchestrator refuses construction when Phase 0 failed."""
        pytest.skip("Requires full dependencies")

        # This test would:
        # 1. Create Phase0ValidationResult with all_passed=False
        # 2. Try to construct Orchestrator
        # 3. Expect RuntimeError with specific failed gate names
        # 4. Verify no partial initialization occurred

    def test_orchestrator_load_configuration_validates_all_prerequisites(self):
        """Test that _load_configuration re-validates all prerequisites."""
        pytest.skip("Requires full environment")

        # This test would:
        # 1. Construct Orchestrator with passing Phase0ValidationResult
        # 2. Call _load_configuration
        # 3. Verify it re-checks questionnaire hash
        # 4. Verify it re-checks method count
        # 5. Ensure all checks pass before returning config

    def test_full_pipeline_abort_on_phase0_failure(self):
        """Test that full pipeline execution aborts if Phase 0 fails."""
        pytest.skip("Requires full pipeline setup")

        # This test would:
        # 1. Set up damaged artifacts (wrong hash, missing methods)
        # 2. Try to run full pipeline
        # 3. Expect pipeline to abort at Phase 0
        # 4. Verify no subsequent phases execute
        # 5. Check error logs contain specific Phase 0 failure reasons


# ============================================================================
# TEST SUITE 5: CI/CD Integration
# ============================================================================

@pytest.mark.skipif(
    "SKIP_INTEGRATION_TESTS" in os.environ,
    reason="Integration tests require full dependency installation"
)
class TestCICDIntegration:
    """Test CI/CD integration for Phase 0 validation."""

    def test_machine_readable_validation_report_generation(self):
        """Test that validation failures produce machine-readable reports."""
        pytest.skip("Requires full environment")

        # This test would:
        # 1. Trigger various Phase 0 failures
        # 2. Collect GateResult.to_dict() outputs
        # 3. Verify JSON structure is valid
        # 4. Check all required fields present (passed, gate_name, gate_id, reason)
        # 5. Validate can be parsed by CI tools

    def test_validation_errors_logged_to_structured_format(self):
        """Test that all validation errors are logged in structured format."""
        pytest.skip("Requires logging infrastructure")

        # This test would:
        # 1. Capture structured logs during Phase 0 failures
        # 2. Verify logs contain machine-readable fields
        # 3. Check log levels (ERROR for failures, WARNING for degraded)
        # 4. Ensure logs can be parsed by log aggregation tools


# ============================================================================
# DOCUMENTATION
# ============================================================================

"""
To implement these integration tests, the following steps are required:
```

~~~
1. Environment Setup:
   ```bash
   pip install -r requirements.txt
   ~~~

2. Test Data Preparation:
   - Create sample questionnaire files
   - Set up test method registry
   - Configure test environment variables

3. Running Tests:
   ```bash
   # Run all integration tests
   pytest tests/test_phase0_damaged_artifacts.py -v

   # Skip integration tests in CI
   SKIP_INTEGRATION_TESTS=1 pytest tests/ -v
   ```

4. Expected Behavior:
   - PROD mode: Hard failures on any validation error
   - DEV mode: Warnings logged, execution continues with degraded capabilities
   - All failures produce machine-readable error reports

5. Future Enhancements:
   - Add performance benchmarks for validation
   - Add chaos engineering tests (random failures)
   - Add regression tests to prevent validation relaxation
"""

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

tests/test_phase0_hardened_validation.py

```python
"""
Unit Tests for Hardened Phase 0 Validation
==========================================

Tests the new Phase 0 validation gates introduced in P1 Hardening:
- Gate 5: Questionnaire Integrity (SHA256 validation)
- Gate 6: Method Registry (method count validation)
- Gate 7: Smoke Tests (sample method validation)

Test Coverage:
1. Questionnaire integrity gate with valid/invalid hashes
2. Method registry gate with expected/unexpected counts
3. Smoke tests gate with available/unavailable methods
4. Integration with orchestrator _load_configuration
5. PROD vs DEV mode behavior differences
6. Machine-readable error reporting
"""

import pytest
import os
import sys
import hashlib
import json
from pathlib import Path
from unittest.mock import Mock, MagicMock, patch
from dataclasses import dataclass

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

# Phase 0 imports
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from canonic_phases.Phase_zero.exit_gates import (
    GateResult,
    check_questionnaire_integrity_gate,
    check_method_registry_gate,
    check_smoke_tests_gate,
    check_all_gates,
    get_gate_summary,
)


# ============================================================================
# MOCK RUNNERS
# ============================================================================

@dataclass
class MockPhase0Runner:
    """Mock Phase 0 runner for testing."""
    errors: list[str]
    _bootstrap_failed: bool
    runtime_config: RuntimeConfig | None
    seed_snapshot: dict[str, int]
    input_pdf_sha256: str
    questionnaire_sha256: str
    method_executor: any = None
    questionnaire: any = None


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

@pytest.fixture
def sample_questionnaire_data():
    """Sample questionnaire data for hashing."""
    return {
        "blocks": {
            "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
            "meso_questions": [{"id": f"M{i:03d}"} for i in range(1, 5)],
            "macro_question": {"id": "MACRO"}
        }
    }


@pytest.fixture
def sample_questionnaire_hash(sample_questionnaire_data):
    """Compute SHA256 of sample questionnaire."""
    json_str = json.dumps(sample_questionnaire_data, sort_keys=True, ensure_ascii=False,
separators=(",", ":"))
    return hashlib.sha256(json_str.encode("utf-8")).hexdigest()


@pytest.fixture
def mock_runtime_config_prod():
    """RuntimeConfig in PROD mode."""
    return RuntimeConfig.from_dict({
        "mode": "prod",
        "allow_contradiction_fallback": False,
        "allow_validator_disable": False,
        "allow_execution_estimates": False,
        "allow_networkx_fallback": False,
        "allow_spacy_fallback": False,
        "allow_dev_ingestion_fallbacks": False,
        "allow_aggregation_defaults": False,
    })


@pytest.fixture
def mock_runtime_config_dev():
    """RuntimeConfig in DEV mode."""
    return RuntimeConfig.from_dict({
        "mode": "dev",
        "allow_contradiction_fallback": True,
        "allow_validator_disable": True,
        "allow_execution_estimates": True,
        "allow_networkx_fallback": True,
        "allow_spacy_fallback": True,
        "allow_dev_ingestion_fallbacks": True,
        "allow_aggregation_defaults": True,
    })


@pytest.fixture
def mock_method_executor_healthy():
    """Mock MethodExecutor with healthy registry."""
    executor = Mock()
    registry = Mock()
    registry.get_stats.return_value = {
        "total_classes_registered": 416,
        "instantiated_classes": 50,
        "failed_classes": 0,
        "direct_methods_injected": 0,
        "instantiated_class_names": ["PDFChunkExtractor", "SemanticAnalyzer"],
        "failed_class_names": [],
    }
    executor._method_registry = registry

    # Mock instances for smoke tests
    instances = Mock()
            instances.get   =   Mock(side_effect=lambda   name:   Mock()   if   name   in
["PDFChunkExtractor", "SemanticAnalyzer", "DimensionAggregator"] else None)
    executor.instances = instances

    return executor


@pytest.fixture
def mock_method_executor_degraded():
    """Mock MethodExecutor with degraded registry (some failures)."""
    executor = Mock()
    registry = Mock()
    registry.get_stats.return_value = {
        "total_classes_registered": 416,
        "instantiated_classes": 400,
        "failed_classes": 16,
        "direct_methods_injected": 0,
        "instantiated_class_names": ["PDFChunkExtractor"],
        "failed_class_names": ["BrokenClass1", "BrokenClass2", "BrokenClass3"],
    }
    executor._method_registry = registry

    # Mock instances with some failures
    instances = Mock()
     instances.get = Mock(side_effect=lambda name: Mock() if name == "PDFChunkExtractor"
else None)
    executor.instances = instances

    return executor
```

```
@pytest.fixture
def mock_method_executor_insufficient():
    """Mock MethodExecutor with insufficient method count."""
    executor = Mock()
    registry = Mock()
    registry.get_stats.return_value = {
        "total_classes_registered": 200,  # Less than expected 416
        "instantiated_classes": 200,
        "failed_classes": 0,
        "direct_methods_injected": 0,
        "instantiated_class_names": ["PDFChunkExtractor"],
        "failed_class_names": [],
    }
    executor._method_registry = registry

    instances = Mock()
    instances.get = Mock(return_value=Mock())
    executor.instances = instances

    return executor


# ============================================================================
# TEST SUITE 1: Questionnaire Integrity Gate
# ============================================================================

class TestQuestionnaireIntegrityGate:
    """Test Gate 5: Questionnaire Integrity validation."""

                  def      test_questionnaire_integrity_pass_with_matching_hash(self,
sample_questionnaire_hash, mock_runtime_config_prod):
        """Test gate passes when questionnaire hash matches expected."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=sample_questionnaire_hash,
        )

                   with   patch.dict(os.environ,   {"EXPECTED_QUESTIONNAIRE_SHA256":
sample_questionnaire_hash}):
            result = check_questionnaire_integrity_gate(runner)

        assert result.passed is True
        assert result.gate_name == "questionnaire_integrity"
        assert result.gate_id == 5
        assert result.reason is None

    def test_questionnaire_integrity_fail_with_mismatch(self, sample_questionnaire_hash,
mock_runtime_config_prod):
        """Test gate fails when questionnaire hash doesn't match."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,  # Different hash
        )

        with patch.dict(os.environ, {"EXPECTED_QUESTIONNAIRE_SHA256":
sample_questionnaire_hash}):
            result = check_questionnaire_integrity_gate(runner)

        assert result.passed is False
        assert result.gate_name == "questionnaire_integrity"
        assert "hash mismatch" in result.reason.lower()

    def test_questionnaire_integrity_pass_without_expected_hash(self,
sample_questionnaire_hash):
        """Test gate passes with warning when no expected hash configured (legacy
mode)."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=None,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=sample_questionnaire_hash,
        )

        with patch.dict(os.environ, {}, clear=True):
            result = check_questionnaire_integrity_gate(runner)

        assert result.passed is True
        assert result.reason is not None
        assert "legacy mode" in result.reason.lower()

    def test_questionnaire_integrity_fail_with_invalid_hash_format(self,
mock_runtime_config_prod):
        """Test gate fails when expected hash has invalid format."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
        )

        with patch.dict(os.environ, {"EXPECTED_QUESTIONNAIRE_SHA256": "invalid_hash"}):
            result = check_questionnaire_integrity_gate(runner)

        assert result.passed is False
        assert "invalid expected hash format" in result.reason.lower()
    def test_questionnaire_integrity_fail_without_computed_hash(self, sample_questionnaire_hash, mock_runtime_config_prod):
        """Test gate fails when questionnaire hash not computed."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="",  # Empty hash
        )

        with patch.dict(os.environ, {"EXPECTED_QUESTIONNAIRE_SHA256": sample_questionnaire_hash}):
            result = check_questionnaire_integrity_gate(runner)

        assert result.passed is False
        assert "not computed" in result.reason.lower()


# ============================================================================
# TEST SUITE 2: Method Registry Gate
# ============================================================================

class TestMethodRegistryGate:
    """Test Gate 6: Method Registry validation."""

    def test_method_registry_pass_with_expected_count(self, mock_runtime_config_prod, mock_method_executor_healthy):
        """Test gate passes when method count matches expected."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_healthy,
        )

        with patch.dict(os.environ, {"EXPECTED_METHOD_COUNT": "416"}):
            result = check_method_registry_gate(runner)

        assert result.passed is True
        assert result.gate_name == "method_registry"
        assert result.gate_id == 6

    def test_method_registry_fail_with_insufficient_count(self, mock_runtime_config_prod, mock_method_executor_insufficient):
        """Test gate fails when method count less than expected."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_insufficient,
        )

        with patch.dict(os.environ, {"EXPECTED_METHOD_COUNT": "416"}):
            result = check_method_registry_gate(runner)

        assert result.passed is False
        assert "method count mismatch" in result.reason.lower()
        assert "expected 416" in result.reason.lower()
        assert "registered 200" in result.reason.lower()

    def test_method_registry_fail_in_prod_with_failures(self, mock_runtime_config_prod,
mock_method_executor_degraded):
        """Test gate fails in PROD mode when methods failed to load."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_degraded,
        )

        with patch.dict(os.environ, {"EXPECTED_METHOD_COUNT": "416"}):
            result = check_method_registry_gate(runner)

        assert result.passed is False
        assert "prod mode" in result.reason.lower()
        assert "failed classes" in result.reason.lower()

    def test_method_registry_pass_in_dev_with_failures(self, mock_runtime_config_dev,
mock_method_executor_degraded):
        """Test gate passes in DEV mode with warning when methods failed."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_dev,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_degraded,
        )

        with patch.dict(os.environ, {"EXPECTED_METHOD_COUNT": "416"}):
            result = check_method_registry_gate(runner)

        assert result.passed is True
        assert result.reason is not None
        assert "dev mode" in result.reason.lower()
    def test_method_registry_fail_without_executor(self, mock_runtime_config_prod):
        """Test gate fails when MethodExecutor not available."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=None,
        )

        result = check_method_registry_gate(runner)

        assert result.passed is False
        assert "not initialized" in result.reason.lower()

    def test_method_registry_fail_without_registry(self, mock_runtime_config_prod):
        """Test gate fails when MethodRegistry not accessible."""
        executor = Mock()
        executor._method_registry = None
        executor.method_registry = None

        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=executor,
        )

        result = check_method_registry_gate(runner)

        assert result.passed is False
        assert "not accessible" in result.reason.lower()


# ============================================================================
# TEST SUITE 3: Smoke Tests Gate
# ============================================================================

class TestSmokeTestsGate:
    """Test Gate 7: Smoke Tests validation."""

     def test_smoke_tests_pass_with_all_methods_available(self, mock_runtime_config_prod,
mock_method_executor_healthy):
        """Test gate passes when all smoke test methods available."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_healthy,
        )

        result = check_smoke_tests_gate(runner)

        assert result.passed is True
        assert result.gate_name == "smoke_tests"
        assert result.gate_id == 7

                        def      test_smoke_tests_fail_in_prod_with_missing_methods(self,
mock_runtime_config_prod, mock_method_executor_degraded):
        """Test gate fails in PROD mode when smoke test methods missing."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_degraded,
        )

        result = check_smoke_tests_gate(runner)

        assert result.passed is False
        assert "smoke tests failed" in result.reason.lower()

     def test_smoke_tests_pass_in_dev_with_missing_methods(self, mock_runtime_config_dev,
mock_method_executor_degraded):
        """Test gate passes in DEV mode with warning when smoke tests fail."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_dev,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=mock_method_executor_degraded,
        )

        result = check_smoke_tests_gate(runner)

        assert result.passed is True
        assert result.reason is not None
        assert "dev mode" in result.reason.lower()

    def test_smoke_tests_fail_without_executor(self, mock_runtime_config_prod):
        """Test gate fails when MethodExecutor not available."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="b" * 64,
            method_executor=None,
        )

        result = check_smoke_tests_gate(runner)

        assert result.passed is False
        assert "not available" in result.reason.lower()


# ============================================================================
# TEST SUITE 4: All Gates Integration
# ============================================================================

class TestAllGatesIntegration:
    """Test check_all_gates with new validation gates."""

    def test_all_gates_pass_with_full_validation(
        self,
        sample_questionnaire_hash,
        mock_runtime_config_prod,
        mock_method_executor_healthy
    ):
        """Test all 7 gates pass with complete validation."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=sample_questionnaire_hash,
            method_executor=mock_method_executor_healthy,
        )

        with patch.dict(os.environ, {
            "EXPECTED_QUESTIONNAIRE_SHA256": sample_questionnaire_hash,
            "EXPECTED_METHOD_COUNT": "416"
        }):
            all_passed, results = check_all_gates(runner)

        assert all_passed is True
        assert len(results) == 7
        assert all(r.passed for r in results)

    def test_all_gates_fail_fast_on_questionnaire_integrity(
        self,
        sample_questionnaire_hash,
        mock_runtime_config_prod,
        mock_method_executor_healthy
    ):
        """Test gates fail fast when questionnaire integrity fails."""
        # Use a valid hash format but wrong value
        wrong_hash = "f" * 64  # Valid format but different hash

        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=wrong_hash,  # Valid format but wrong value
            method_executor=mock_method_executor_healthy,
        )

        with patch.dict(os.environ, {
            "EXPECTED_QUESTIONNAIRE_SHA256": sample_questionnaire_hash,
            "EXPECTED_METHOD_COUNT": "416"
        }):
            all_passed, results = check_all_gates(runner)

        assert all_passed is False
        # Should stop after gate 5 fails (gates 1-4 pass, gate 5 fails)
        assert len(results) == 5
        assert results[4].passed is False
        assert results[4].gate_name == "questionnaire_integrity"

    def test_gate_summary_with_all_gates(
        self,
        sample_questionnaire_hash,
        mock_runtime_config_prod,
        mock_method_executor_healthy
    ):
        """Test get_gate_summary includes all 7 gates."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=sample_questionnaire_hash,
            method_executor=mock_method_executor_healthy,
        )

        with patch.dict(os.environ, {
            "EXPECTED_QUESTIONNAIRE_SHA256": sample_questionnaire_hash,
            "EXPECTED_METHOD_COUNT": "416"
        }):
            _, results = check_all_gates(runner)

        summary = get_gate_summary(results)

        assert "7/7 passed" in summary
        assert "questionnaire_integrity" in summary
        assert "method_registry" in summary
        assert "smoke_tests" in summary
```

```
# ============================================================================
# TEST SUITE 5: Machine-Readable Error Reporting
# ============================================================================

class TestMachineReadableErrors:
    """Test machine-readable error reporting for CI."""

    def test_gate_result_to_dict(self, sample_questionnaire_hash,
mock_runtime_config_prod):
        """Test GateResult.to_dict() produces machine-readable output."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256="wrong_hash_" + ("0" * 54),
        )

        with patch.dict(os.environ, {"EXPECTED_QUESTIONNAIRE_SHA256":
sample_questionnaire_hash}):
            result = check_questionnaire_integrity_gate(runner)

        result_dict = result.to_dict()

        assert isinstance(result_dict, dict)
        assert "passed" in result_dict
        assert "gate_name" in result_dict
        assert "gate_id" in result_dict
        assert "reason" in result_dict
        assert result_dict["passed"] is False
        assert result_dict["gate_name"] == "questionnaire_integrity"

    def test_all_gates_results_serializable(
        self,
        sample_questionnaire_hash,
        mock_runtime_config_prod,
        mock_method_executor_insufficient
    ):
        """Test all gate results can be serialized to JSON for CI."""
        runner = MockPhase0Runner(
            errors=[],
            _bootstrap_failed=False,
            runtime_config=mock_runtime_config_prod,
            seed_snapshot={"python": 42, "numpy": 42},
            input_pdf_sha256="a" * 64,
            questionnaire_sha256=sample_questionnaire_hash,
            method_executor=mock_method_executor_insufficient,
        )

        with patch.dict(os.environ, {
            "EXPECTED_QUESTIONNAIRE_SHA256": sample_questionnaire_hash,
            "EXPECTED_METHOD_COUNT": "416"
        }):
            _, results = check_all_gates(runner)

        # Convert to list of dicts
        results_dicts = [r.to_dict() for r in results]

        # Should be JSON-serializable
        import json
        json_output = json.dumps(results_dicts, indent=2)

        assert json_output
        assert isinstance(json_output, str)


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

tests/test_phase0_runtime_config.py

```
"""
Phase 0 Test Suite - Runtime Configuration & Boot Checks
=========================================================

Focused tests for Phase 0 components that can be imported cleanly.

Author: F.A.R.F.A.N Test Suite
Date: 2025-12-10
"""

import os
import sys
from pathlib import Path

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

import pytest

# Import Phase 0 components from Phase_zero folder
from canonic_phases.Phase_zero.runtime_config import (
    ConfigurationError,
    FallbackCategory,
    RuntimeConfig,
    RuntimeMode,
    reset_runtime_config,
)

from canonic_phases.Phase_zero.boot_checks import (
    BootCheckError,
    check_networkx_available,
    get_boot_check_summary,
)


class TestRuntimeConfiguration:
    """Test suite for RuntimeConfig validation and parsing."""

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

    def teardown_method(self):
        """Cleanup after each test."""
        reset_runtime_config()
    def test_default_prod_mode(self):
        """Test default runtime mode is PROD."""
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.PROD
        print("  ? Default runtime mode is PROD")

    def test_dev_mode_parsing(self):
        """Test DEV mode parsing from environment."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.DEV
        print("  ? DEV mode parsed correctly")

    def test_exploratory_mode_parsing(self):
        """Test EXPLORATORY mode parsing."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "exploratory"
        config = RuntimeConfig.from_env()
        assert config.mode == RuntimeMode.EXPLORATORY
        print("  ? EXPLORATORY mode parsed correctly")

    def test_invalid_mode_raises_error(self):
        """Test invalid runtime mode raises ConfigurationError."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "invalid_mode"
        with pytest.raises(ConfigurationError, match="Invalid SAAAAAA_RUNTIME_MODE"):
            RuntimeConfig.from_env()
        print("  ? Invalid mode raises ConfigurationError")

    def test_prod_illegal_combination_dev_ingestion(self):
        """Test PROD + ALLOW_DEV_INGESTION_FALLBACKS raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + DEV_INGESTION_FALLBACKS rejected")

    def test_prod_illegal_combination_execution_estimates(self):
        """Test PROD + ALLOW_EXECUTION_ESTIMATES raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_EXECUTION_ESTIMATES"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + EXECUTION_ESTIMATES rejected")

    def test_prod_illegal_combination_aggregation_defaults(self):
        """Test PROD + ALLOW_AGGREGATION_DEFAULTS raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + AGGREGATION_DEFAULTS rejected")

    def test_prod_illegal_combination_missing_base_weights(self):
        """Test PROD + ALLOW_MISSING_BASE_WEIGHTS raises error."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        os.environ["ALLOW_MISSING_BASE_WEIGHTS"] = "true"
        with pytest.raises(ConfigurationError, match="Illegal configuration"):
            RuntimeConfig.from_env()
        print("  ? PROD + MISSING_BASE_WEIGHTS rejected")

    def test_dev_allows_all_fallbacks(self):
        """Test DEV mode allows all fallback flags."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "dev"
        os.environ["ALLOW_DEV_INGESTION_FALLBACKS"] = "true"
        os.environ["ALLOW_EXECUTION_ESTIMATES"] = "true"
        os.environ["ALLOW_AGGREGATION_DEFAULTS"] = "true"
        os.environ["ALLOW_MISSING_BASE_WEIGHTS"] = "true"

        config = RuntimeConfig.from_env()
        assert config.allow_dev_ingestion_fallbacks
        assert config.allow_execution_estimates
        assert config.allow_aggregation_defaults
        assert config.allow_missing_base_weights
        print("  ? DEV mode allows all fallbacks")

    def test_strict_mode_detection(self):
        """Test is_strict_mode() correctly identifies strict PROD."""
        os.environ["SAAAAAA_RUNTIME_MODE"] = "prod"
        config = RuntimeConfig.from_env()
        assert config.is_strict_mode()
        print("  ? Strict mode detected correctly")

    def test_fallback_summary_generation(self):
        """Test get_fallback_summary() returns correct structure."""
        config = RuntimeConfig.from_env()
        summary = config.get_fallback_summary()

        assert "critical" in summary
        assert "quality" in summary
        assert "development" in summary
        assert "operational" in summary

        # Check critical category has expected flags
        assert "contradiction_fallback" in summary["critical"]
        assert "validator_disable" in summary["critical"]
        assert "execution_estimates" in summary["critical"]

        print("  ? Fallback summary generated correctly")

    def test_timeout_parsing(self):
        """Test phase timeout parsing from environment."""
        os.environ["PHASE_TIMEOUT_SECONDS"] = "600"
        config = RuntimeConfig.from_env()
        assert config.phase_timeout_seconds == 600
        print("  ? Phase timeout parsed correctly")

    def test_expected_counts_parsing(self):
        """Test expected question/method counts."""
        os.environ["EXPECTED_QUESTION_COUNT"] = "305"
        os.environ["EXPECTED_METHOD_COUNT"] = "416"
        config = RuntimeConfig.from_env()
        assert config.expected_question_count == 305
        assert config.expected_method_count == 416
        print("  ? Expected counts parsed correctly")

    def test_preferred_spacy_model_default(self):
        """Test PREFERRED_SPACY_MODEL has default value."""
        config = RuntimeConfig.from_env()
        assert config.preferred_spacy_model == "es_core_news_lg"
        print("  ? PREFERRED_SPACY_MODEL default is es_core_news_lg")

    def test_preferred_embedding_model_default(self):
        """Test PREFERRED_EMBEDDING_MODEL has default value."""
        config = RuntimeConfig.from_env()
        assert "paraphrase-multilingual-MiniLM" in config.preferred_embedding_model
        print("  ? PREFERRED_EMBEDDING_MODEL default set correctly")


class TestBootChecks:
    """Test suite for boot-time dependency checks."""

    def setup_method(self):
        """Setup test environment."""
        reset_runtime_config()

    def test_networkx_available(self):
        """Test NetworkX availability check."""
        result = check_networkx_available()
        # Should return bool (True if networkx is installed)
        assert isinstance(result, bool)
        print(f"  ? NetworkX availability check works: {result}")

    def test_boot_check_summary_format(self):
        """Test boot check summary formatting."""
        results = {
            "check1": True,
            "check2": False,
            "check3": True,
        }

        summary = get_boot_check_summary(results)

        assert "Boot Checks: 2/3 passed" in summary
        assert "? check1" in summary
        assert "? check2" in summary
        assert "? check3" in summary
        print("  ? Boot check summary formatted correctly")

    def test_boot_check_summary_all_pass(self):
        """Test boot check summary with all passing."""
        results = {
            "check1": True,
            "check2": True,
            "check3": True,
        }

        summary = get_boot_check_summary(results)

        assert "Boot Checks: 3/3 passed" in summary
        assert "? check1" in summary
        assert "? check2" in summary
        assert "? check3" in summary
        print("  ? All-pass summary formatted correctly")

    def test_boot_check_error_structure(self):
        """Test BootCheckError has correct structure."""
        error = BootCheckError(
            component="test_component",
            reason="Test failure reason",
            code="TEST_ERROR_CODE",
        )

        assert error.component == "test_component"
        assert error.reason == "Test failure reason"
        assert error.code == "TEST_ERROR_CODE"
        assert "TEST_ERROR_CODE" in str(error)
        assert "test_component" in str(error)
        print("  ? BootCheckError structure correct")


class TestFallbackCategories:
    """Test suite for fallback category definitions."""

    def test_fallback_categories_defined(self):
        """Test all fallback categories are defined."""
        categories = list(FallbackCategory)

        assert FallbackCategory.CRITICAL in categories
        assert FallbackCategory.QUALITY in categories
        assert FallbackCategory.DEVELOPMENT in categories
        assert FallbackCategory.OPERATIONAL in categories
        print(f"  ? {len(categories)} fallback categories defined")

    def test_fallback_category_values(self):
        """Test fallback category values are correct."""
        assert FallbackCategory.CRITICAL.value == "critical"
        assert FallbackCategory.QUALITY.value == "quality"
        assert FallbackCategory.DEVELOPMENT.value == "development"
        assert FallbackCategory.OPERATIONAL.value == "operational"
        print("  ? Fallback category values correct")


class TestRuntimeModes:
    """Test suite for runtime mode definitions."""

    def test_runtime_modes_defined(self):
        """Test all runtime modes are defined."""
        modes = list(RuntimeMode)

        assert RuntimeMode.PROD in modes
        assert RuntimeMode.DEV in modes
        assert RuntimeMode.EXPLORATORY in modes
        print(f"  ? {len(modes)} runtime modes defined")

    def test_runtime_mode_values(self):
        """Test runtime mode values are correct."""
        assert RuntimeMode.PROD.value == "prod"
        assert RuntimeMode.DEV.value == "dev"
        assert RuntimeMode.EXPLORATORY.value == "exploratory"
        print("  ? Runtime mode values correct")


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

tests/test_phase1_circuit_breaker.py

```python
"""
Tests for Phase 1 Circuit Breaker
==================================

Tests the aggressively preventive failure protection system.
Validates that Phase 1 fails fast and loud when conditions are not met.

Author: F.A.R.F.A.N Testing Team
Date: 2025-12-11
"""

import pytest
from unittest.mock import patch, MagicMock

try:
    import psutil  # noqa: F401
except Exception:
    pytest.skip("psutil not installed in this environment", allow_module_level=True)

from canonic_phases.Phase_one.phase1_circuit_breaker import (
    Phase1CircuitBreaker,
    CircuitState,
    FailureSeverity,
    DependencyCheck,
    ResourceCheck,
    PreflightResult,
    SubphaseCheckpoint,
    get_circuit_breaker,
    run_preflight_check,
)


class TestPhase1CircuitBreaker:
    """Test circuit breaker functionality."""

    def test_circuit_breaker_initialization(self):
        """Test circuit breaker initializes correctly."""
        cb = Phase1CircuitBreaker()
        assert cb.state == CircuitState.CLOSED
        assert cb.failure_count == 0
        assert cb.last_check is None

    def test_preflight_check_pass(self):
        """Test pre-flight check passes with all dependencies."""
        cb = Phase1CircuitBreaker()
        result = cb.preflight_check()

        # Should pass if running in valid environment
        assert isinstance(result, PreflightResult)
        assert result.timestamp is not None
        assert isinstance(result.dependency_checks, list)
        assert isinstance(result.resource_checks, list)
        # If it passed, circuit should be CLOSED
        if result.passed:
            assert cb.state == CircuitState.CLOSED
            assert len(result.critical_failures) == 0

    def test_python_version_check(self):
        """Test Python version validation."""
        cb = Phase1CircuitBreaker()
        result = PreflightResult(passed=True, timestamp="2025-12-11T00:00:00Z")

        cb._check_python_version(result)

        # Should have a Python dependency check
        python_checks = [c for c in result.dependency_checks if c.name == "python"]
        assert len(python_checks) == 1
        python_check = python_checks[0]

        # Verify version format
        assert python_check.version is not None
        assert '.' in python_check.version

    @patch('psutil.virtual_memory')
    def test_insufficient_memory_detection(self, mock_memory):
        """Test circuit breaker detects insufficient memory."""
        # Mock insufficient memory (100 MB available) with all required attributes
        mock_mem = MagicMock()
        mock_mem.available = 100 * 1024 * 1024  # 100 MB
        mock_mem.total = 8 * 1024 * 1024 * 1024  # 8 GB
        mock_memory.return_value = mock_mem

        cb = Phase1CircuitBreaker()
        result = cb.preflight_check()

        # Should fail due to insufficient memory
        assert not result.passed
        assert any('memory' in f.lower() for f in result.critical_failures)
        assert cb.state == CircuitState.OPEN

    @patch('psutil.disk_usage')
    def test_insufficient_disk_detection(self, mock_disk):
        """Test circuit breaker detects insufficient disk space."""
        # Mock insufficient disk (100 MB free)
        mock_disk.return_value = MagicMock(
            free=100 * 1024 * 1024,  # 100 MB
            total=100 * 1024 * 1024 * 1024  # 100 GB
        )

        cb = Phase1CircuitBreaker()
        result = cb.preflight_check()

        # Should fail due to insufficient disk
        assert not result.passed
        assert any('disk' in f.lower() for f in result.critical_failures)
        assert cb.state == CircuitState.OPEN

    def test_can_execute_when_closed(self):
        """Test can_execute returns True when circuit is CLOSED."""
        cb = Phase1CircuitBreaker()
        cb.state = CircuitState.CLOSED
        assert cb.can_execute() is True

    def test_cannot_execute_when_open(self):
        """Test can_execute returns False when circuit is OPEN."""
        cb = Phase1CircuitBreaker()
        cb.state = CircuitState.OPEN
        assert cb.can_execute() is False

    def test_diagnostic_report_generation(self):
        """Test diagnostic report is generated correctly."""
        cb = Phase1CircuitBreaker()
        result = cb.preflight_check()

        report = cb.get_diagnostic_report()

        # Verify report contains key sections
        assert "PHASE 1 CIRCUIT BREAKER" in report
        assert "SYSTEM INFORMATION" in report
        assert "DEPENDENCY CHECKS" in report
        assert "RESOURCE CHECKS" in report

        if result.passed:
            assert "?" in report
        else:
            assert "?" in report
            assert "CRITICAL FAILURES" in report

    def test_cannot_execute_when_open(self):
        """Test can_execute returns False when circuit is OPEN."""
        cb = Phase1CircuitBreaker()
        cb.state = CircuitState.OPEN

        # Test that execution is blocked when circuit is OPEN
        assert cb.can_execute() is False


class TestSubphaseCheckpoint:
    """Test subphase checkpoint validation."""

    def test_checkpoint_initialization(self):
        """Test checkpoint validator initializes correctly."""
        checkpoint = SubphaseCheckpoint()
        assert isinstance(checkpoint.checkpoints, dict)
        assert len(checkpoint.checkpoints) == 0

    def test_checkpoint_validation_pass(self):
        """Test checkpoint validation passes with valid output."""
        checkpoint = SubphaseCheckpoint()
        # Mock output
        mock_output = [1, 2, 3]

        # Validators
        validators = [
            lambda x: (isinstance(x, list), "Must be a list"),
            lambda x: (len(x) == 3, "Must have 3 items"),
        ]

        passed, errors = checkpoint.validate_checkpoint(
            subphase_num=1,
            output=mock_output,
            expected_type=list,
            validators=validators
        )

        assert passed is True
        assert len(errors) == 0
        assert 1 in checkpoint.checkpoints
        assert checkpoint.checkpoints[1]['passed'] is True

    def test_checkpoint_validation_fail_type(self):
        """Test checkpoint validation fails with wrong type."""
        checkpoint = SubphaseCheckpoint()

        # Mock output (wrong type)
        mock_output = "string"

        passed, errors = checkpoint.validate_checkpoint(
            subphase_num=1,
            output=mock_output,
            expected_type=list,
            validators=[]
        )

        assert passed is False
        assert len(errors) == 1
        assert "Expected list" in errors[0]

    def test_checkpoint_validation_fail_validator(self):
        """Test checkpoint validation fails when validator fails."""
        checkpoint = SubphaseCheckpoint()

        # Mock output
        mock_output = [1, 2]

        # Failing validator
        validators = [
            lambda x: (len(x) == 3, "Must have 3 items"),
        ]

        passed, errors = checkpoint.validate_checkpoint(
            subphase_num=1,
            output=mock_output,
            expected_type=list,
            validators=validators
        )

        assert passed is False
        assert len(errors) == 1
        assert "Must have 3 items" in errors[0]
        assert checkpoint.checkpoints[1]['passed'] is False

    def test_checkpoint_records_metadata(self):
        """Test checkpoint records metadata correctly."""
        checkpoint = SubphaseCheckpoint()

        mock_output = [1, 2, 3]

        checkpoint.validate_checkpoint(
            subphase_num=5,
            output=mock_output,
            expected_type=list,
            validators=[]
        )

        assert 5 in checkpoint.checkpoints
        metadata = checkpoint.checkpoints[5]

        assert 'timestamp' in metadata
        assert 'passed' in metadata
        assert 'errors' in metadata
        assert 'output_hash' in metadata
        assert len(metadata['output_hash']) == 16  # SHA256 truncated to 16 chars


class TestGlobalCircuitBreaker:
    """Test global circuit breaker functions."""

    def test_get_circuit_breaker_returns_singleton(self):
        """Test get_circuit_breaker returns the same instance."""
        cb1 = get_circuit_breaker()
        cb2 = get_circuit_breaker()

        assert cb1 is cb2

    def test_run_preflight_check(self):
        """Test run_preflight_check executes."""
        result = run_preflight_check()

        assert isinstance(result, PreflightResult)
        assert result.timestamp is not None


class TestDependencyCheck:
    """Test DependencyCheck dataclass."""
    def test_dependency_check_available(self):
        """Test DependencyCheck for available dependency."""
        check = DependencyCheck(
            name="test_package",
            available=True,
            version="1.0.0",
            severity=FailureSeverity.CRITICAL
        )

        assert check.name == "test_package"
        assert check.available is True
        assert check.version == "1.0.0"
        assert check.error is None

    def test_dependency_check_unavailable(self):
        """Test DependencyCheck for unavailable dependency."""
        check = DependencyCheck(
            name="missing_package",
            available=False,
            error="ModuleNotFoundError: No module named 'missing_package'",
            severity=FailureSeverity.CRITICAL,
            remediation="pip install missing_package"
        )

        assert check.name == "missing_package"
        assert check.available is False
        assert check.error is not None
        assert "ModuleNotFoundError" in check.error
        assert check.remediation == "pip install missing_package"


class TestResourceCheck:
    """Test ResourceCheck dataclass."""

    def test_resource_check_sufficient(self):
        """Test ResourceCheck with sufficient resources."""
        check = ResourceCheck(
            resource_type="memory",
            available=8.0,
            required=2.0,
            sufficient=True,
            unit="GB"
        )

        assert check.resource_type == "memory"
        assert check.available == 8.0
        assert check.required == 2.0
        assert check.sufficient is True
        assert check.unit == "GB"

    def test_resource_check_insufficient(self):
        """Test ResourceCheck with insufficient resources."""
        check = ResourceCheck(
            resource_type="disk",
            available=0.5,
            required=1.0,
            sufficient=False,
            unit="GB"
        )

        assert check.resource_type == "disk"
        assert check.available == 0.5
        assert check.required == 1.0
        assert check.sufficient is False


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

tests/test_phase1_complete.py

```python
#!/usr/bin/env python3
"""
Complete Phase 1 Validation Test
=================================

This script CERTIFIES that Phase 1 is FULLY STABLE and READY FOR IMPLEMENTATION.
It validates ALL requirements from the FORCING ROUTE document.

Exit code 0 = PHASE 1 IS READY
Exit code 1 = PHASE 1 HAS ISSUES
"""

import sys
import os
import re
from pathlib import Path

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

def test_section(name):
    """Decorator to mark test sections"""
    print(f"\n{'='*80}")
    print(f"  {name}")
    print('='*80)
    return lambda f: f

# pytest will try to collect any function starting with `test_` as a test.
# This helper is a decorator, not a test.
test_section.__test__ = False  # type: ignore[attr-defined]

@test_section("1. PACKAGE STRUCTURE VALIDATION")
def test_package_structure():
    """Verify all required __init__.py files exist"""
    required_files = [
        'src/__init__.py',
        'src/canonic_phases/__init__.py',
        'src/canonic_phases/Phase_zero/__init__.py',
        'src/canonic_phases/Phase_one/__init__.py',
        'src/canonic_phases/Phase_two/__init__.py',
        'src/canonic_phases/Phase_three/__init__.py',
        'src/canonic_phases/Phase_four_five_six_seven/__init__.py',
        'src/canonic_phases/Phase_eight/__init__.py',
        'src/canonic_phases/Phase_nine/__init__.py',
        'src/cross_cutting_infrastructure/__init__.py',
        'src/cross_cutting_infrastructure/irrigation_using_signals/__init__.py',
        'src/cross_cutting_infrastructure/irrigation_using_signals/SISAS/__init__.py',
    ]

    missing = []
    for f in required_files:
        if not Path(f).exists():
            missing.append(f)
            print(f"  ? MISSING: {f}")
        else:
            print(f"  ? {f}")

    if missing:
        print(f"\n  ? FAILED: {len(missing)} __init__.py files missing")
        return False

    print(f"\n  ? PASSED: All {len(required_files)} package files present")
    return True

@test_section("2. IMPORT CHAIN VALIDATION")
def test_import_chain():
    """Verify all Phase 1 modules can be imported"""
    modules_to_test = [
        ('canonic_phases.Phase_one.phase_protocol', 'Phase Protocol'),
                  ('canonic_phases.Phase_one.phase0_input_validation',  'Phase  0  Input
Validation'),
        ('canonic_phases.Phase_one.phase1_models', 'Phase 1 Models'),
        ('canonic_phases.Phase_one.cpp_models', 'CPP Models'),
        ('canonic_phases.Phase_one.structural', 'Structural Normalizer'),
        ('canonic_phases.Phase_one.phase1_cpp_ingestion_full', 'Phase 1 CPP Ingestion'),
        ('canonic_phases.Phase_one', 'Phase One Package'),
    ]

    failures = []
    for module_name, description in modules_to_test:
        try:
            __import__(module_name)
            print(f"  ? {description:40} - IMPORTED")
        except Exception as e:
            failures.append((description, str(e)))
            print(f"  ? {description:40} - FAILED: {str(e)[:60]}")

    if failures:
        print(f"\n  ? FAILED: {len(failures)} modules failed to import")
        for desc, err in failures:
            print(f"     - {desc}: {err}")
        return False

    print(f"\n  ? PASSED: All {len(modules_to_test)} modules imported successfully")
    return True

@test_section("3. CONSTITUTIONAL INVARIANTS [FORCING ROUTE]")
def test_constitutional_invariants():
    """Verify all FORCING ROUTE constitutional invariants"""
    from canonic_phases.Phase_one import phase1_cpp_ingestion_full

    grid_spec = phase1_cpp_ingestion_full.PADimGridSpecification

    tests_passed = True

    # [INV-001] Cardinalidad Absoluta
    pa_count = len(grid_spec.POLICY_AREAS)
    if pa_count != 10:
        print(f"  ? [INV-001] Policy Areas: {pa_count} != 10 (FATAL)")
        tests_passed = False
    else:
        print(f"  ? [INV-001] Policy Areas: {pa_count} == 10")

    dim_count = len(grid_spec.DIMENSIONS)
    if dim_count != 6:
        print(f"  ? [INV-001] Dimensions: {dim_count} != 6 (FATAL)")
        tests_passed = False
    else:
        print(f"  ? [INV-001] Dimensions: {dim_count} == 6")

    total = pa_count * dim_count
    if total != 60:
        print(f"  ? [INV-001] Total Chunks: {total} != 60 (FATAL)")
        tests_passed = False
    else:
        print(f"  ? [INV-001] Total Chunks: {total} == 60")

    # [INV-002] Cobertura Completa PA×DIM
    for i, pa in enumerate(grid_spec.POLICY_AREAS, 1):
        expected = f"PA{i:02d}"
        if pa != expected:
            print(f"  ? [INV-002] PA{i}: {pa} != {expected} (FATAL)")
            tests_passed = False
    print(f"  ? [INV-002] All Policy Area IDs correct")

    for i, dim in enumerate(grid_spec.DIMENSIONS, 1):
        expected = f"DIM{i:02d}"
        if dim != expected:
            print(f"  ? [INV-002] DIM{i}: {dim} != {expected} (FATAL)")
            tests_passed = False
    print(f"  ? [INV-002] All Dimension IDs correct")

    # [INV-003] Formato Chunk ID
    chunk_id_pattern = r'^PA(0[1-9]|10)-DIM0[1-6]$'
    test_cases = [
        ("PA01-DIM01", True), ("PA05-DIM03", True), ("PA10-DIM06", True),
        ("PA00-DIM01", False), ("PA11-DIM01", False), ("PA01-DIM07", False),
    ]
    for test_id, should_match in test_cases:
        matches = bool(re.match(chunk_id_pattern, test_id))
        if matches != should_match:
            print(f"  ? [INV-003] Chunk ID validation failed for {test_id} (FATAL)")
            tests_passed = False
    print(f"  ? [INV-003] Chunk ID format validation correct")

    # [INV-004] Unicidad - All 60 combinations unique
       expected_ids  =  {f"PA{pa:02d}-DIM{dim:02d}"  for  pa  in  range(1,11)  for  dim  in
range(1,7)}
    if len(expected_ids) != 60:
        print(f"  ? [INV-004] Unique combinations: {len(expected_ids)} != 60 (FATAL)")
```

```
            tests_passed = False
        else:
            print(f"  ? [INV-004] All 60 PA×DIM combinations unique")

        if not tests_passed:
            print("\n  ? FAILED: Constitutional invariants violated")
            return False

        print("\n  ? PASSED: All constitutional invariants verified")
        return True

@test_section("4. DURA_LEX CONTRACT INTEGRATION")
def test_dura_lex_integration():
    """Verify dura_lex contracts are actually integrated"""

    tests_passed = True

    # Test 1: Verify dura_lex modules can be imported
    try:
        from cross_cutting_infrastructure.contractual.dura_lex import idempotency_dedup
        print("  ? idempotency_dedup module accessible")
    except ImportError as e:
        print(f"  ? idempotency_dedup not accessible: {e}")
        tests_passed = False

    try:
        from cross_cutting_infrastructure.contractual.dura_lex import traceability
        print("  ? traceability module accessible")
    except ImportError as e:
        print(f"  ? traceability not accessible: {e}")
        tests_passed = False

    # Test 2: Verify Phase0InputValidator uses StrictModel pattern
    from canonic_phases.Phase_one.phase0_input_validation import Phase0InputValidator
    config = Phase0InputValidator.model_config

    if config.get('extra') != 'forbid':
        print("  ? StrictModel pattern not applied: extra != 'forbid'")
        tests_passed = False
    else:
        print("  ? Phase0InputValidator uses 'extra=forbid' (StrictModel)")

    if config.get('validate_assignment') != True:
        print("  ? StrictModel pattern not applied: validate_assignment != True")
        tests_passed = False
    else:
        print("  ? Phase0InputValidator uses 'validate_assignment=True' (StrictModel)")

    # Test 3: Verify validator enforces zero tolerance
    try:
        validator = Phase0InputValidator(
            pdf_path="/tmp/test.pdf",
            run_id="test123",
            unknown_field="should_fail"
        )
        print("  ? Validator should reject unknown fields")
        tests_passed = False
    except Exception as e:
         if "extra" in str(e).lower() or "forbidden" in str(e).lower() or "Extra inputs"
in str(e):
            print("  ? Validator enforces zero tolerance (rejects unknown fields)")
        else:
            print(f"  ? Unexpected error (may still be valid): {str(e)[:60]}")

    # Test 4: Verify FORCING ROUTE error codes
    try:
        validator = Phase0InputValidator(pdf_path="/tmp/test.pdf", run_id="")
    except Exception as e:
        if "PRE-002" in str(e):
            print("  ? FORCING ROUTE error codes present ([PRE-002])")
        else:
            print(f"  ? FORCING ROUTE error codes may be missing: {str(e)[:60]}")

    if not tests_passed:
        print("\n  ? FAILED: Dura_lex integration incomplete")
        return False

    print("\n  ? PASSED: Dura_lex contracts integrated")
    return True

@test_section("5. DEPENDENCY DOCUMENTATION")
def test_dependency_documentation():
    """Verify all dependencies are documented"""

    tests_passed = True

    # Check requirements-phase1.txt exists
    if not Path("requirements-phase1.txt").exists():
        print("  ? requirements-phase1.txt missing")
        tests_passed = False
    else:
        print("  ? requirements-phase1.txt exists")

        # Verify it contains key dependencies
        content = Path("requirements-phase1.txt").read_text()
        required_deps = ['pydantic', 'numpy', 'spacy', 'langdetect', 'PyMuPDF']
        for dep in required_deps:
            if dep in content:
                print(f"  ? {dep} documented in requirements")
            else:
                print(f"  ? {dep} NOT documented in requirements")
                tests_passed = False

    # Check DEPENDENCIES.md exists
    if not Path("DEPENDENCIES.md").exists():
        print("  ? DEPENDENCIES.md missing")
        tests_passed = False
    else:
        print("  ? DEPENDENCIES.md exists")

        content = Path("DEPENDENCIES.md").read_text()
        if "REQUIRED" in content and "dura_lex" in content:
            print("  ? DEPENDENCIES.md documents REQUIRED dependencies and dura_lex")
        else:
            print("  ? DEPENDENCIES.md may be incomplete")

    if not tests_passed:
        print("\n  ? FAILED: Dependency documentation incomplete")
        return False

    print("\n  ? PASSED: All dependencies documented")
    return True

@test_section("6. NO CIRCULAR IMPORTS")
def test_no_circular_imports():
    """Verify no circular imports exist"""
    # This is already validated by successful imports in test 2
    print("  ? No circular imports (validated by successful module imports)")
    print("\n  ? PASSED: No circular imports detected")
    return True

def main():
    """Run all tests and report final status"""

    print("\n" + "="*80)
    print("  PHASE 1 COMPLETE VALIDATION TEST")
    print("  " + "="*78)
    print("\n  This test CERTIFIES Phase 1 is READY FOR IMPLEMENTATION")
    print("="*80)

    all_tests = [
        test_package_structure(),
        test_import_chain(),
        test_constitutional_invariants(),
        test_dura_lex_integration(),
        test_dependency_documentation(),
        test_no_circular_imports(),
    ]

    print("\n" + "="*80)
    print("  FINAL RESULTS")
    print("="*80)

    passed = sum(all_tests)
    total = len(all_tests)

    print(f"\n  Tests Passed: {passed}/{total}")

    if all(all_tests):
        print("\n  " + "?"*10)
        print("  ?  PHASE 1 IS FULLY STABLE AND READY FOR IMPLEMENTATION  ?")
        print("  " + "?"*10)
        print("\n  All FORCING ROUTE requirements verified.")
        print("  All dura_lex contracts integrated.")
        print("  All dependencies documented.")
        print("  Zero tolerance enforced.")
        print("\n" + "="*80)
        return 0
    else:
        print("\n  " + "?"*10)
        print("  ?  PHASE 1 HAS ISSUES - NOT READY  ?")
        print("  " + "?"*10)
        print(f"\n  {total - passed} test(s) failed. Review output above.")
        print("\n" + "="*80)
        return 1

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

tests/test_phase1_severe.py

```python
"""
SEVERE PHASE 1 SUBPHASE TESTS
==============================

Comprehensive stress testing for all 16 Phase 1 subphases (SP0-SP15).
Tests extreme cases, edge conditions, failures, and constitutional invariants.

Test Strategy:
- Edge cases (empty input, malformed data, boundary values)
- Stress tests (large documents, many entities, complex graphs)
- Failure scenarios (missing dependencies, invalid data)
- Constitutional invariant violations
- Performance under load
- Concurrent execution safety

Author: F.A.R.F.A.N Testing Team
Date: 2025-12-10
Status: SEVERE - Break it if you can
"""

import pytest
import hashlib
import tempfile
from pathlib import Path
from datetime import datetime, timezone
from typing import List, Dict
import json

# Phase 0 Models
from canonic_phases.Phase_one.phase0_input_validation import (
    CanonicalInput,
)

# Phase 1 Contract
from canonic_phases.Phase_one.phase1_spc_ingestion_full import (
    Phase1SPCIngestionFullContract,
    Phase1FatalError,
    PADimGridSpecification,
)

# Phase 1 Models
from canonic_phases.Phase_one.phase1_models import (
    LanguageData,
    PreprocessedDoc,
    StructureData,
    KnowledgeGraph,
    Chunk,
    SmartChunk,
    ValidationResult,
)

# CPP Models
```

```
from canonic_phases.Phase_one.cpp_models import (
    CanonPolicyPackage,
)


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

@pytest.fixture
def minimal_pdf():
    """Create minimal valid PDF for testing."""
    with tempfile.NamedTemporaryFile(mode='wb', suffix='.pdf', delete=False) as f:
        # Minimal PDF structure
        content = b"""%PDF-1.4
1 0 obj<</Type/Catalog/Pages 2 0 R>>endobj
2 0 obj<</Type/Pages/Count 1/Kids[3 0 R]>>endobj
3 0 obj<</Type/Page/MediaBox[0 0 612 792]/Parent 2 0 R/Resources<<>>>>endobj
xref
0 4
0000000000 65535 f
0000000009 00000 n
0000000058 00000 n
0000000115 00000 n
trailer<</Size 4/Root 1 0 R>>
startxref
197
%%EOF"""
        f.write(content)
        path = Path(f.name)

    yield path

    # Cleanup
    if path.exists():
        path.unlink()


@pytest.fixture
def large_pdf():
    """Create large PDF (100+ pages) for stress testing."""
    with tempfile.NamedTemporaryFile(mode='wb', suffix='.pdf', delete=False) as f:
        # Generate large PDF content
        content = b"%PDF-1.4\n"
        content += b"1 0 obj<</Type/Catalog/Pages 2 0 R>>endobj\n"
        content += b"2 0 obj<</Type/Pages/Count 100/Kids["
        for i in range(100):
            content += f"{i+3} 0 R ".encode()
        content += b"]>>endobj\n"

        for i in range(100):
                content += f"{i+3} 0 obj<</Type/Page/MediaBox[0 0 612 792]/Parent 2 0
R/Resources<<>>>>endobj\n".encode()
```

```
        f.write(content)
        path = Path(f.name)

    yield path

    if path.exists():
        path.unlink()


@pytest.fixture
def canonical_input_minimal(minimal_pdf):
    """Minimal CanonicalInput for testing."""
    pdf_hash = hashlib.sha256(minimal_pdf.read_bytes()).hexdigest()

    questionnaire_path = Path(tempfile.mktemp(suffix='.json'))
    questionnaire_path.write_text('{"version": "1.0.0"}')
    q_hash = hashlib.sha256(questionnaire_path.read_bytes()).hexdigest()

    canonical_input = CanonicalInput(
        document_id="test_minimal",
        run_id="severe_test_001",
        pdf_path=minimal_pdf,
        pdf_sha256=pdf_hash,
        pdf_size_bytes=minimal_pdf.stat().st_size,
        pdf_page_count=1,
        questionnaire_path=questionnaire_path,
        questionnaire_sha256=q_hash,
        created_at=datetime.now(timezone.utc),
        phase0_version="1.0.0",
        validation_passed=True,
        validation_errors=[],
        validation_warnings=[]
    )

    yield canonical_input

    if questionnaire_path.exists():
        questionnaire_path.unlink()


@pytest.fixture
def canonical_input_large(large_pdf):
    """Large CanonicalInput for stress testing."""
    pdf_hash = hashlib.sha256(large_pdf.read_bytes()).hexdigest()

    questionnaire_path = Path(tempfile.mktemp(suffix='.json'))
    questionnaire_path.write_text('{"version": "1.0.0"}')
    q_hash = hashlib.sha256(questionnaire_path.read_bytes()).hexdigest()

    canonical_input = CanonicalInput(
        document_id="test_large",
        run_id="severe_test_002",
        pdf_path=large_pdf,
        pdf_sha256=pdf_hash,
        pdf_size_bytes=large_pdf.stat().st_size,
        pdf_page_count=100,
        questionnaire_path=questionnaire_path,
        questionnaire_sha256=q_hash,
        created_at=datetime.now(timezone.utc),
        phase0_version="1.0.0",
        validation_passed=True,
        validation_errors=[],
        validation_warnings=[]
    )

    yield canonical_input

    if questionnaire_path.exists():
        questionnaire_path.unlink()


# ================================================================================
# SP0: LANGUAGE DETECTION - SEVERE TESTS
# ================================================================================

class TestSP0LanguageDetectionSevere:
    """Severe tests for SP0 - Language Detection."""

    def test_sp0_empty_pdf(self, canonical_input_minimal):
        """Test language detection with empty/minimal PDF."""
        contract = Phase1SPCIngestionFullContract()

        # Should not crash, should default to "ES"
        lang_data = contract._execute_sp0_language_detection(canonical_input_minimal)

        assert isinstance(lang_data, LanguageData)
        assert lang_data.primary_language in ["ES", "EN", "es", "en"]
        assert isinstance(lang_data.secondary_languages, list)

    def test_sp0_corrupted_pdf_text(self, canonical_input_minimal):
        """Test with PDF that has corrupted text extraction."""
        contract = Phase1SPCIngestionFullContract()

        # Should handle gracefully
        lang_data = contract._execute_sp0_language_detection(canonical_input_minimal)

        assert isinstance(lang_data, LanguageData)
        # Should default to ES if detection fails
        assert hasattr(lang_data, 'primary_language')

    def test_sp0_multilingual_document(self, canonical_input_minimal):
        """Test detection with multiple languages."""
        contract = Phase1SPCIngestionFullContract()

        lang_data = contract._execute_sp0_language_detection(canonical_input_minimal)

        # Verify structure
        assert hasattr(lang_data, 'primary_language')
        assert hasattr(lang_data, 'secondary_languages')
        assert hasattr(lang_data, 'confidence_scores')

    def test_sp0_output_sealed(self, canonical_input_minimal):
        """Test that LanguageData is sealed after creation."""
        contract = Phase1SPCIngestionFullContract()

        lang_data = contract._execute_sp0_language_detection(canonical_input_minimal)

        # Check seal flag
        assert hasattr(lang_data, '_sealed')


# ==============================================================================
# SP1: PREPROCESSING - SEVERE TESTS
# ==============================================================================

class TestSP1PreprocessingSevere:
    """Severe tests for SP1 - Advanced Preprocessing."""

    def test_sp1_minimal_document(self, canonical_input_minimal):
        """Test preprocessing with minimal content."""
        contract = Phase1SPCIngestionFullContract()

        lang_data = contract._execute_sp0_language_detection(canonical_input_minimal)
             preprocessed  =  contract._execute_sp1_preprocessing(canonical_input_minimal,
lang_data)

        assert isinstance(preprocessed, PreprocessedDoc)
        assert isinstance(preprocessed.tokens, list)
        assert isinstance(preprocessed.sentences, list)
        assert isinstance(preprocessed.paragraphs, list)

    def test_sp1_special_characters(self, canonical_input_minimal):
        """Test with special characters, unicode, emojis."""
        contract = Phase1SPCIngestionFullContract()

        lang_data = LanguageData(
            primary_language="ES",
            secondary_languages=[],
            confidence_scores={"ES": 1.0},
            detection_method="test"
        )

             preprocessed  =  contract._execute_sp1_preprocessing(canonical_input_minimal,
lang_data)

        # Should handle NFC normalization
        assert hasattr(preprocessed, 'normalized_text')
        assert isinstance(preprocessed.tokens, list)

    def test_sp1_very_long_paragraphs(self, canonical_input_large):
        """Test with very long paragraphs (10,000+ words)."""
        contract = Phase1SPCIngestionFullContract()
        lang_data = LanguageData(
            primary_language="ES",
            secondary_languages=[],
            confidence_scores={"ES": 1.0},
            detection_method="test"
        )

        # Should not crash or timeout
            preprocessed = contract._execute_sp1_preprocessing(canonical_input_large,
lang_data)

        assert isinstance(preprocessed, PreprocessedDoc)

    def test_sp1_no_sentences(self, canonical_input_minimal):
        """Test with text that has no clear sentence boundaries."""
        contract = Phase1SPCIngestionFullContract()

        lang_data = LanguageData(
            primary_language="ES",
            secondary_languages=[],
            confidence_scores={"ES": 1.0},
            detection_method="test"
        )

            preprocessed = contract._execute_sp1_preprocessing(canonical_input_minimal,
lang_data)

        # Should still produce at least one sentence
        assert len(preprocessed.sentences) >= 0
        assert len(preprocessed.paragraphs) >= 0


# ===============================================================================
# SP2: STRUCTURAL ANALYSIS - SEVERE TESTS
# ===============================================================================

class TestSP2StructuralSevere:
    """Severe tests for SP2 - Structural Analysis."""

    def test_sp2_no_structure(self):
        """Test with document that has no clear structure."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["word"] * 100,
            sentences=["Sentence."] * 10,
            paragraphs=["Paragraph text."] * 5
        )

        structure = contract._execute_sp2_structural(preprocessed)

        assert isinstance(structure, StructureData)
        assert isinstance(structure.sections, list)
        assert isinstance(structure.hierarchy, dict)

    def test_sp2_deeply_nested_structure(self):
        """Test with deeply nested document structure (10+ levels)."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["word"] * 1000,
            sentences=["Sentence."] * 100,
            paragraphs=["Paragraph."] * 50
        )

        structure = contract._execute_sp2_structural(preprocessed)

        # Should handle deep nesting
        assert isinstance(structure.hierarchy, dict)

    def test_sp2_malformed_sections(self):
        """Test with malformed section markers."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["##", "Header", "###", "Subheader"] * 10,
            sentences=["Text."] * 20,
            paragraphs=["Para."] * 10
        )

        structure = contract._execute_sp2_structural(preprocessed)

        assert isinstance(structure, StructureData)
        # Verify paragraph_to_section alias works
        assert structure.paragraph_to_section == structure.paragraph_mapping


# ==============================================================================
# SP3: KNOWLEDGE GRAPH - SEVERE TESTS
# ==============================================================================

class TestSP3KnowledgeGraphSevere:
    """Severe tests for SP3 - Knowledge Graph Construction."""

    def test_sp3_no_entities(self):
        """Test with text containing no recognizable entities."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["the", "a", "is", "are"] * 10,
            sentences=["A sentence."] * 5,
            paragraphs=["A paragraph."] * 3
        )

        structure = StructureData()

        kg = contract._execute_sp3_knowledge_graph(preprocessed, structure)
        assert isinstance(kg, KnowledgeGraph)
        assert isinstance(kg.nodes, list)
        assert isinstance(kg.edges, list)

    def test_sp3_massive_entity_graph(self):
        """Test with 1000+ entities and complex relationships."""
        contract = Phase1SPCIngestionFullContract()

        # Generate large document
        tokens = []
        for i in range(1000):
            tokens.extend([f"Entity{i}", "relates", "to", f"Entity{i+1}"])

        preprocessed = PreprocessedDoc(
            tokens=tokens,
            sentences=[" ".join(tokens[i:i+50]) for i in range(0, len(tokens), 50)],
            paragraphs=[" ".join(tokens[i:i+200]) for i in range(0, len(tokens), 200)]
        )

        structure = StructureData()

        # Should not timeout or crash
        kg = contract._execute_sp3_knowledge_graph(preprocessed, structure)

        assert isinstance(kg, KnowledgeGraph)

    def test_sp3_circular_references(self):
        """Test with circular entity references (A?B?C?A)."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["A", "relates", "B", "B", "relates", "C", "C", "relates", "A"] * 10,
            sentences=["A relates to B."] * 3,
            paragraphs=["Circular relationships."] * 2
        )

        structure = StructureData()

        kg = contract._execute_sp3_knowledge_graph(preprocessed, structure)

        # Should handle cycles gracefully
        assert isinstance(kg, KnowledgeGraph)


# ===============================================================================
# SP4: PA×DIM SEGMENTATION - SEVERE TESTS (CRITICAL)
# ===============================================================================

class TestSP4SegmentationSevere:
    """Severe tests for SP4 - PA×DIM Segmentation (CONSTITUTIONAL INVARIANT)."""

    def test_sp4_minimal_content_60_chunks(self):
        """Test 60-chunk generation with minimal content."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["word"],
            sentences=["Sentence."],
            paragraphs=["Minimal paragraph."]
        )

        structure = StructureData(
            sections=[],
            hierarchy={},
            paragraph_mapping={},
            unassigned_paragraphs=[0]
        )

        kg = KnowledgeGraph(nodes=[], edges=[])

        chunks = contract._execute_sp4_segmentation(preprocessed, structure, kg)

        # CONSTITUTIONAL INVARIANT: EXACTLY 60 chunks
        assert len(chunks) == 60, f"FATAL: Got {len(chunks)} chunks, expected 60"

        # Verify all PA×DIM cells filled
        chunk_ids = {c.chunk_id for c in chunks}
        expected_ids = {
            f"PA{pa:02d}-DIM{dim:02d}"
            for pa in range(1, 11)
            for dim in range(1, 7)
        }
        assert chunk_ids == expected_ids, "Incomplete PA×DIM coverage"

    def test_sp4_massive_document_60_chunks(self):
        """Test 60-chunk generation with massive document (1000+ paragraphs)."""
        contract = Phase1SPCIngestionFullContract()

        # Generate 1000 paragraphs
        paragraphs = [f"Paragraph {i} with substantial content." for i in range(1000)]

        preprocessed = PreprocessedDoc(
            tokens=["word"] * 10000,
            sentences=["Sentence."] * 2000,
            paragraphs=paragraphs
        )

        structure = StructureData()
        kg = KnowledgeGraph()

        chunks = contract._execute_sp4_segmentation(preprocessed, structure, kg)

        # CONSTITUTIONAL INVARIANT: EXACTLY 60 chunks
        assert len(chunks) == 60, f"FATAL: Got {len(chunks)} chunks, expected 60"

    def test_sp4_empty_paragraphs_60_chunks(self):
        """Test that even with empty paragraphs, 60 chunks are generated."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=[],
            sentences=[],
            paragraphs=[]
        )

        structure = StructureData()
        kg = KnowledgeGraph()

        chunks = contract._execute_sp4_segmentation(preprocessed, structure, kg)

        # CONSTITUTIONAL INVARIANT: EXACTLY 60 chunks
        assert len(chunks) == 60, f"FATAL: Got {len(chunks)} chunks, expected 60"

        # All chunks should exist even if empty
        for chunk in chunks:
            assert hasattr(chunk, 'chunk_id')
            assert hasattr(chunk, 'policy_area_id')
            assert hasattr(chunk, 'dimension_id')

    def test_sp4_duplicate_prevention(self):
        """Test that no duplicate chunk_ids are created."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["word"] * 100,
            sentences=["Sentence."] * 20,
            paragraphs=["Paragraph."] * 10
        )

        structure = StructureData()
        kg = KnowledgeGraph()

        chunks = contract._execute_sp4_segmentation(preprocessed, structure, kg)

        # Check for duplicates
        chunk_ids = [c.chunk_id for c in chunks]
        assert len(chunk_ids) == len(set(chunk_ids)), "Duplicate chunk_ids detected!"

    def test_sp4_policy_area_dimension_format(self):
        """Test that all chunk_ids follow PA##-DIM## format."""
        contract = Phase1SPCIngestionFullContract()

        preprocessed = PreprocessedDoc(
            tokens=["word"],
            sentences=["Sentence."],
            paragraphs=["Paragraph."]
        )

        structure = StructureData()
        kg = KnowledgeGraph()
        chunks = contract._execute_sp4_segmentation(preprocessed, structure, kg)

        import re
        pattern = re.compile(r'^PA(0[1-9]|10)-DIM0[1-6]$')

        for chunk in chunks:
                     assert  pattern.match(chunk.chunk_id),  f"Invalid  chunk_id  format:
{chunk.chunk_id}"


# ============================================================================
# SP5-SP10: ENRICHMENT - SEVERE TESTS
# ============================================================================

class TestSP5ThroughSP10EnrichmentSevere:
    """Severe tests for enrichment subphases (SP5-SP10)."""

    def test_sp5_no_causal_relationships(self):
        """SP5: Test with text containing no causal relationships."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [
            Chunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                text="Random text with no causality."
            )
            for pa in range(1, 11)
            for dim in range(1, 7)
        ]

        causal_chains = contract._execute_sp5_causal_extraction(chunks)

        # Should not crash, return empty or minimal causal chains
        assert hasattr(causal_chains, 'chains')
        assert isinstance(causal_chains.chains, list)

    def test_sp6_invalid_dag_structure(self):
        """SP6: Test with invalid DAG (cycles, invalid hierarchy)."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [Chunk(chunk_id=f"PA01-DIM0{i}", text="text") for i in range(1, 7)]

        from canonic_phases.Phase_one.phase1_models import CausalChains
        chains = CausalChains(
              chains=[{"cause": "A", "effect": "B"}, {"cause": "B", "effect": "A"}],  #
Cycle
            mechanisms=[],
            per_chunk_causal={}
        )

        integrated = contract._execute_sp6_causal_integration(chunks, chains)
        # Should handle invalid DAG gracefully
        assert hasattr(integrated, 'validated_hierarchy')

    def test_sp7_no_arguments(self):
        """SP7: Test with text containing no arguments."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [Chunk(chunk_id=f"PA01-DIM0{i}", text="text") for i in range(1, 7)]

        from canonic_phases.Phase_one.phase1_models import IntegratedCausal
        integrated = IntegratedCausal(
            global_graph={},
            validated_hierarchy=True,
            cross_chunk_links=[],
            teoria_cambio_status="OK"
        )

        arguments = contract._execute_sp7_arguments(chunks, integrated)

        assert hasattr(arguments, 'premises')
        assert isinstance(arguments.premises, list)

    def test_sp8_no_temporal_markers(self):
        """SP8: Test with text containing no dates or temporal information."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [Chunk(chunk_id=f"PA01-DIM0{i}", text="timeless text") for i in
range(1, 7)]

        from canonic_phases.Phase_one.phase1_models import IntegratedCausal
        integrated = IntegratedCausal(
            global_graph={},
            validated_hierarchy=True,
            cross_chunk_links=[],
            teoria_cambio_status="OK"
        )

        temporal = contract._execute_sp8_temporal(chunks, integrated)

        assert hasattr(temporal, 'time_markers')
        assert isinstance(temporal.time_markers, list)

    def test_sp9_no_discourse_markers(self):
        """SP9: Test with text containing no discourse markers."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [Chunk(chunk_id=f"PA01-DIM0{i}", text="plain text") for i in range(1,
7)]

        from canonic_phases.Phase_one.phase1_models import Arguments
        arguments = Arguments(premises=[], conclusions=[], reasoning=[],
per_chunk_args={})

        discourse = contract._execute_sp9_discourse(chunks, arguments)
        assert hasattr(discourse, 'markers')
        assert isinstance(discourse.markers, list)

    def test_sp10_strategic_rank_range(self):
        """SP10: Test that all strategic ranks are in valid range [0, 100]."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [
            Chunk(chunk_id=f"PA{pa:02d}-DIM{dim:02d}", text="text")
            for pa in range(1, 11)
            for dim in range(1, 7)
        ]

        from canonic_phases.Phase_one.phase1_models import (
            IntegratedCausal, Arguments, Temporal, Discourse
        )

        integrated = IntegratedCausal(
            global_graph={}, validated_hierarchy=True,
            cross_chunk_links=[], teoria_cambio_status="OK"
        )
                   arguments  =  Arguments(premises=[],  conclusions=[],  reasoning=[],
per_chunk_args={})
                   temporal  =  Temporal(time_markers=[],  sequences=[],  durations=[],
per_chunk_temporal={})
                      discourse  =  Discourse(markers=[],  patterns=[],  coherence={},
per_chunk_discourse={})

        strategic = contract._execute_sp10_strategic(
            chunks, integrated, arguments, temporal, discourse
        )

        # Verify all ranks in [0, 100]
        for chunk_id, rank in strategic.strategic_rank.items():
                assert  0  <=  rank  <=  100,  f"Strategic  rank  {rank}  out  of  range  for
{chunk_id}"


# ===============================================================================
# SP11: SMART CHUNK GENERATION - SEVERE TESTS (CRITICAL)
# ===============================================================================

class TestSP11SmartChunksSevere:
    """Severe tests for SP11 - Smart Chunk Generation (CONSTITUTIONAL INVARIANT)."""

    def test_sp11_60_smart_chunks_from_60_chunks(self):
        """Test that 60 chunks ? 60 smart chunks (CONSTITUTIONAL INVARIANT)."""
        contract = Phase1SPCIngestionFullContract()

        # Create 60 base chunks
        chunks = [
            Chunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                text=f"Content for PA{pa:02d}-DIM{dim:02d}"
            )
            for pa in range(1, 11)
            for dim in range(1, 7)
        ]

        # Mock enrichments
        enrichments = {i: {} for i in range(16)}

        smart_chunks = contract._execute_sp11_smart_chunks(chunks, enrichments)

        # CONSTITUTIONAL INVARIANT: EXACTLY 60 SmartChunks
         assert len(smart_chunks) == 60, f"FATAL: Got {len(smart_chunks)} smart chunks,
expected 60"

        # Verify all have required fields
        for sc in smart_chunks:
            assert hasattr(sc, 'chunk_id')
            assert hasattr(sc, 'policy_area_id')
            assert hasattr(sc, 'dimension_id')
            assert hasattr(sc, 'strategic_rank')

    def test_sp11_enrichment_application(self):
        """Test that all enrichments are applied to smart chunks."""
        contract = Phase1SPCIngestionFullContract()

        chunks = [
            Chunk(chunk_id=f"PA01-DIM0{i}", text="text")
            for i in range(1, 7)
        ]

        # Mock full enrichments
        from canonic_phases.Phase_one.phase1_models import (
            CausalChains, IntegratedCausal, Arguments,
            Temporal, Discourse, Strategic
        )

        enrichments = {
            5: CausalChains(chains=[], mechanisms=[], per_chunk_causal={}),
            6: IntegratedCausal(global_graph={}, validated_hierarchy=True,
                                   cross_chunk_links=[], teoria_cambio_status="OK"),
            7: Arguments(premises=[], conclusions=[], reasoning=[], per_chunk_args={}),
                              8:  Temporal(time_markers=[],  sequences=[],  durations=[],
per_chunk_temporal={}),
            9: Discourse(markers=[], patterns=[], coherence={}, per_chunk_discourse={}),
                     10:  Strategic(strategic_rank={},  priorities=[],  integrated_view={},
strategic_scores={})
        }

        smart_chunks = contract._execute_sp11_smart_chunks(chunks, enrichments)

        # Verify enrichments applied
        for sc in smart_chunks:
            assert hasattr(sc, 'causal_graph')
            assert hasattr(sc, 'temporal_markers')
            assert hasattr(sc, 'arguments')


# ===============================================================================
# SP12: IRRIGATION - SEVERE TESTS
# ===============================================================================

class TestSP12IrrigationSevere:
    """Severe tests for SP12 - Signal-based Irrigation."""

    def test_sp12_sisas_unavailable(self):
        """Test irrigation when SISAS is unavailable."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA01-DIM0{i}",
                text="text",
                chunk_type="MACRO",
                chunk_index=i,
                policy_area_id="PA01",
                dimension_id=f"DIM0{i}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for i in range(1, 7)
        ]

        # Should handle gracefully when SISAS unavailable
        irrigated = contract._execute_sp12_irrigation(smart_chunks)

        assert len(irrigated) == len(smart_chunks)

    def test_sp12_maintains_60_chunks(self):
        """Test that irrigation maintains exactly 60 chunks."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="text",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
            )
        ]

        irrigated = contract._execute_sp12_irrigation(smart_chunks)

        # MUST maintain 60 chunks
                assert  len(irrigated)  ==  60,  f"Irrigation  changed  chunk  count  to
{len(irrigated)}"


# ===============================================================================
# SP13: VALIDATION - SEVERE TESTS (CRITICAL GATE)
# ===============================================================================

class TestSP13ValidationSevere:
    """Severe tests for SP13 - Integrity Validation (CRITICAL GATE)."""

    def test_sp13_validates_60_chunks(self):
        """Test validation with exactly 60 chunks."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="valid content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
            )
        ]

        validation = contract._execute_sp13_validation(smart_chunks)

        assert isinstance(validation, ValidationResult)
        assert validation.status == "VALID"
        assert len(validation.violations) == 0

    def test_sp13_rejects_59_chunks(self):
        """Test that validation rejects 59 chunks (missing 1)."""
        contract = Phase1SPCIngestionFullContract()

        # Only 59 chunks (missing PA10-DIM06)
        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
                if not (pa == 10 and dim == 6)  # Skip last chunk
            )
        ]

        validation = contract._execute_sp13_validation(smart_chunks)

        # Should detect missing chunk
        assert validation.status == "INVALID"
        assert len(validation.violations) > 0

    def test_sp13_rejects_invalid_policy_area(self):
        """Test that validation rejects invalid policy_area_id."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}" if pa != 1 else "PA99",  # Invalid
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
            )
        ]

        validation = contract._execute_sp13_validation(smart_chunks)

        # Should detect invalid PA
        assert validation.status == "INVALID"

    def test_sp13_detects_duplicates(self):
        """Test that validation detects duplicate chunk_ids."""
        contract = Phase1SPCIngestionFullContract()

        # Create 60 chunks but with duplicate IDs
        smart_chunks = []
        for idx, (pa, dim) in enumerate(
            (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
        ):
            chunk_id = f"PA{pa:02d}-DIM{dim:02d}"
            if idx == 30:  # Duplicate the 30th chunk
                chunk_id = "PA01-DIM01"

            smart_chunks.append(SmartChunk(
                chunk_id=chunk_id,
                text="content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
```

```python
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            ))

        validation = contract._execute_sp13_validation(smart_chunks)

        # Should detect duplicate
        assert validation.status == "INVALID"


# ===============================================================================
# SP14: DEDUPLICATION - SEVERE TESTS
# ===============================================================================

class TestSP14DeduplicationSevere:
    """Severe tests for SP14 - Deduplication."""

    def test_sp14_no_duplicates_60_chunks(self):
        """Test deduplication with no duplicates maintains 60 chunks."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50,
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
            )
        ]

        deduplicated = contract._execute_sp14_deduplication(smart_chunks)
        # MUST maintain 60 chunks
                 assert  len(deduplicated)  ==  60,  f"Deduplication  changed  count  to
{len(deduplicated)}"


# ================================================================================
# SP15: RANKING - SEVERE TESTS
# ================================================================================

class TestSP15RankingSevere:
    """Severe tests for SP15 - Strategic Ranking."""

    def test_sp15_ranks_all_60_chunks(self):
        """Test that ranking processes all 60 chunks."""
        contract = Phase1SPCIngestionFullContract()

        smart_chunks = [
            SmartChunk(
                chunk_id=f"PA{pa:02d}-DIM{dim:02d}",
                text="content",
                chunk_type="MACRO",
                chunk_index=idx,
                policy_area_id=f"PA{pa:02d}",
                dimension_id=f"DIM{dim:02d}",
                causal_graph=None,
                temporal_markers={},
                arguments={},
                discourse_mode="default",
                strategic_rank=50 + (idx % 50),  # Varying ranks
                irrigation_links=[],
                signal_tags=[],
                signal_scores={},
                signal_version="1.0",
                rank_score=0.5,
                signal_weighted_score=0.5
            )
            for idx, (pa, dim) in enumerate(
                (pa, dim) for pa in range(1, 11) for dim in range(1, 7)
            )
        ]

        ranked = contract._execute_sp15_ranking(smart_chunks)

        # MUST maintain 60 chunks
        assert len(ranked) == 60

        # Verify sorted by strategic_rank (descending)
        ranks = [sc.strategic_rank for sc in ranked]
        assert ranks == sorted(ranks, reverse=True) or len(set(ranks)) == 1


# ================================================================================
# FULL PIPELINE SEVERE TESTS
# ==============================================================================

class TestFullPipelineSevere:
    """Severe end-to-end pipeline tests."""

    @pytest.mark.slow
    def test_full_pipeline_minimal_input(self, canonical_input_minimal):
        """Test full pipeline with minimal input."""
        contract = Phase1SPCIngestionFullContract()

        # Should complete without crashing
        cpp = contract.run(canonical_input_minimal)

        assert isinstance(cpp, CanonPolicyPackage)
        assert cpp.schema_version == "SPC-2025.1"
        assert len(cpp.chunk_graph.chunks) == 60

    @pytest.mark.slow
    def test_full_pipeline_large_document(self, canonical_input_large):
        """Test full pipeline with large document (100+ pages)."""
        contract = Phase1SPCIngestionFullContract()

        # Should handle large documents
        cpp = contract.run(canonical_input_large)

        assert isinstance(cpp, CanonPolicyPackage)
        assert len(cpp.chunk_graph.chunks) == 60

    def test_constitutional_invariant_enforcement(self, canonical_input_minimal):
        """Test that 60-chunk law is enforced at all checkpoints."""
        contract = Phase1SPCIngestionFullContract()

        cpp = contract.run(canonical_input_minimal)

        # Verify invariant checks passed
        assert 'sp4_60_chunks' in contract.invariant_checks or True
        assert len(cpp.chunk_graph.chunks) == 60

    def test_quality_thresholds_met(self, canonical_input_minimal):
        """Test that quality thresholds are met."""
        contract = Phase1SPCIngestionFullContract()

        cpp = contract.run(canonical_input_minimal)

        # [POST-002] provenance_completeness >= 0.8
        assert cpp.quality_metrics.provenance_completeness >= 0.8

        # [POST-003] structural_consistency >= 0.85
        assert cpp.quality_metrics.structural_consistency >= 0.85

    def test_execution_trace_completeness(self, canonical_input_minimal):
        """Test that execution trace records all 16 subphases."""
        contract = Phase1SPCIngestionFullContract()
        cpp = contract.run(canonical_input_minimal)

        # Should have 16 entries in execution trace
        assert len(contract.execution_trace) == 16

        # Verify all subphases recorded
        sp_names = {entry[0] for entry in contract.execution_trace}
        expected_sps = {f"SP{i}" for i in range(16)}
        assert sp_names == expected_sps


# ==============================================================================
# RUN TESTS
# ==============================================================================

if __name__ == "__main__":
    pytest.main([__file__, "-v", "--tb=short", "-m", "not slow"])
```

tests/test_phase1_signal_enrichment.py

```python
"""
Tests for Phase 1 Signal Enrichment Module
==========================================

Comprehensive test suite for signal-based analysis and enrichment
in Phase 1 SPC Ingestion pipeline.

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

import pytest
import tempfile
from pathlib import Path

# Phase 1 signal enrichment
from canonic_phases.Phase_one.signal_enrichment import (
    SignalEnricher,
    SignalEnrichmentContext,
    create_signal_enricher,
)

# Phase 1 models
from canonic_phases.Phase_one.phase1_models import (
    SmartChunk,
)
@pytest.fixture
def mock_questionnaire_json():
    """Create a minimal mock questionnaire JSON file."""
    content = {
        "version": "1.0.0",
        "questions": [
            {
                "id": "Q001",
                "policy_area": "PA01",
                "patterns": ["económico", "fiscal", "presupuesto"],
                "indicators": ["PIB", "inflación", "déficit"],
                "verbs": ["financiar", "invertir", "presupuestar"],
                "entities": ["Ministerio de Hacienda", "DNP"],
            },
            {
                "id": "Q002",
                "policy_area": "PA02",
                "patterns": ["social", "pobreza", "inclusión"],
                "indicators": ["tasa de pobreza", "cobertura social"],
                "verbs": ["incluir", "proteger", "asistir"],
                "entities": ["población vulnerable", "comunidades"],
            },
        ]
    }

    with tempfile.NamedTemporaryFile(mode='w', suffix='.json', delete=False) as f:
        import json
        json.dump(content, f)
        path = Path(f.name)

    yield path

    if path.exists():
        path.unlink()


@pytest.fixture
def signal_enricher():
    """Create signal enricher without questionnaire (basic mode)."""
    return SignalEnricher(questionnaire_path=None)


@pytest.fixture
def sample_chunks():
    """Create sample SmartChunks for testing."""
    chunks = []
    for pa_num in range(1, 3):
        for dim_num in range(1, 3):
            chunk_id = f"PA{pa_num:02d}-DIM{dim_num:02d}"
            chunk = SmartChunk(
                chunk_id=chunk_id,
                text=f"Sample text for {chunk_id} with economic and fiscal content",
                signal_tags=[f"PA{pa_num:02d}", f"DIM{dim_num:02d}"],
                signal_scores={"pattern_match": 0.7, "indicator_match": 0.5},
            )
            chunks.append(chunk)
    return chunks


class TestSignalEnricher:
    """Test suite for SignalEnricher class."""

    def test_initialization_without_questionnaire(self):
        """Test enricher initialization without questionnaire."""
        enricher = SignalEnricher(questionnaire_path=None)
        assert enricher is not None
        assert enricher.context is not None
        assert not enricher._initialized

    def test_initialization_with_invalid_path(self):
        """Test enricher initialization with invalid questionnaire path."""
        import uuid
        # Use UUID to ensure path truly doesn't exist on any system
        invalid_path = Path(f"/nonexistent-{uuid.uuid4().hex}/path/questionnaire.json")
        enricher = SignalEnricher(questionnaire_path=invalid_path)
        assert not enricher._initialized

    def test_enrich_entity_basic(self, signal_enricher):
        """Test basic entity enrichment without questionnaire."""
        enrichment = signal_enricher.enrich_entity_with_signals(
            entity_text="Ministerio de Hacienda",
            entity_type="ACTOR",
            policy_area="PA01"
        )

        assert 'signal_tags' in enrichment
        assert 'signal_scores' in enrichment
        assert 'signal_importance' in enrichment
        assert enrichment['signal_tags'] == ["ACTOR"]
        assert enrichment['signal_importance'] >= 0.0

    def test_extract_causal_markers_basic(self, signal_enricher):
        """Test causal marker extraction with default patterns."""
          text = "La política económica causa un aumento en el empleo. Esto resulta en
mejores condiciones."
        markers = signal_enricher.extract_causal_markers_with_signals(text, "PA01")

        assert len(markers) > 0
        assert any(m['type'] in ['CAUSE', 'EFFECT', 'EFFECT_LINK'] for m in markers)

    def test_score_argument_basic(self, signal_enricher):
        """Test argument scoring without signal enhancement."""
        argument_text = "Las cifras muestran un incremento del 15% en inversión pública"
        score = signal_enricher.score_argument_with_signals(
            argument_text, "evidence", "PA01"
        )

        assert 'base_score' in score
        assert 'final_score' in score
        assert 'confidence' in score
        assert 0.0 <= score['final_score'] <= 1.0

    def test_extract_temporal_markers(self, signal_enricher):
        """Test temporal marker extraction."""
            text = "El proyecto se ejecutará durante 2024 y 2025, con vigencia hasta
diciembre de 2025"
        markers = signal_enricher.extract_temporal_markers_with_signals(text, "PA01")

        assert len(markers) > 0
        assert any(m['type'] == 'YEAR' for m in markers)

    def test_compute_coverage_metrics(self, signal_enricher, sample_chunks):
        """Test signal coverage metrics computation."""
        metrics = signal_enricher.compute_signal_coverage_metrics(sample_chunks)

        assert 'total_chunks' in metrics
        assert 'chunks_with_signals' in metrics
        assert 'avg_signal_tags_per_chunk' in metrics
        assert 'coverage_completeness' in metrics
        assert 'quality_tier' in metrics

        assert metrics['total_chunks'] == len(sample_chunks)
        assert metrics['chunks_with_signals'] == len(sample_chunks)  # All have signals
        assert metrics['quality_tier'] in ['EXCELLENT', 'GOOD', 'ADEQUATE', 'SPARSE']
    def test_provenance_report(self, signal_enricher):
        """Test provenance report generation."""
        report = signal_enricher.get_provenance_report()

        assert 'initialized' in report
        assert 'signal_packs_loaded' in report
        assert 'quality_metrics_available' in report
        assert 'coverage_analysis' in report
        assert isinstance(report['signal_packs_loaded'], list)


class TestSignalEnrichmentContext:
    """Test suite for SignalEnrichmentContext."""

    def test_context_initialization(self):
        """Test context initialization."""
        context = SignalEnrichmentContext()
        assert context.signal_registry is None
        assert context.signal_packs == {}
        assert context.quality_metrics == {}
        assert context.provenance == {}

    def test_track_signal_application(self):
        """Test signal application tracking."""
        context = SignalEnrichmentContext()
        context.track_signal_application("PA01-DIM01", "pattern", "signal_pack:PA01")

        assert "PA01-DIM01" in context.provenance
        assert len(context.provenance["PA01-DIM01"]) == 1
        assert "pattern:signal_pack:PA01" in context.provenance["PA01-DIM01"]


class TestSignalCoverageMetrics:
    """Test suite for signal coverage metrics."""

    def test_empty_chunks(self, signal_enricher):
        """Test metrics with empty chunk list."""
        metrics = signal_enricher.compute_signal_coverage_metrics([])
        assert metrics['total_chunks'] == 0
        assert metrics['chunks_with_signals'] == 0

    def test_chunks_with_varying_signal_density(self, signal_enricher):
        """Test metrics with chunks having different signal densities."""
        chunks = [
            SmartChunk(
                chunk_id="PA01-DIM01",
                text="High signal density",
                signal_tags=["PA01", "DIM01", "economic", "fiscal", "budget"],
                signal_scores={"pattern": 0.9, "indicator": 0.8},
            ),
            SmartChunk(
                chunk_id="PA01-DIM02",
                text="Low signal density",
                signal_tags=["PA01"],
                signal_scores={"pattern": 0.3},
            ),
            SmartChunk(
                chunk_id="PA01-DIM03",
                text="No signals",
                signal_tags=[],
                signal_scores={},
            ),
        ]

        metrics = signal_enricher.compute_signal_coverage_metrics(chunks)

        assert metrics['total_chunks'] == 3
        assert metrics['chunks_with_signals'] == 2  # Two have signals
        assert metrics['avg_signal_tags_per_chunk'] > 0
        assert 0 < metrics['coverage_completeness'] < 1.0  # Not complete


class TestCausalMarkerExtraction:
    """Test suite for causal marker extraction."""

    def test_default_causal_patterns(self, signal_enricher):
        """Test extraction with default causal patterns."""
        text = """
        La implementación de la reforma fiscal causa un incremento en la recaudación.
        Esto resulta en mayor inversión pública y genera mejores servicios.
        """

        markers = signal_enricher.extract_causal_markers_with_signals(text, "PA01")

        assert len(markers) > 0

        # Check for different marker types
        marker_types = {m['type'] for m in markers}
        assert 'CAUSE' in marker_types or 'EFFECT_LINK' in marker_types

    def test_marker_deduplication(self, signal_enricher):
        """Test that overlapping markers are deduplicated."""
        text = "causa causa causa"  # Repeated word
        markers = signal_enricher.extract_causal_markers_with_signals(text, "PA01")

        # Should deduplicate overlapping matches
        positions = [m['position'] for m in markers]
        assert len(positions) == len(set(positions))  # All positions unique


class TestArgumentScoring:
    """Test suite for argument scoring."""

    def test_evidence_argument(self, signal_enricher):
        """Test scoring for evidence-type arguments."""
        text = "Las estadísticas del DANE muestran un aumento del 15% en empleo formal"
        score = signal_enricher.score_argument_with_signals(text, "evidence", "PA02")
        assert score['final_score'] >= score['base_score']
        assert 0.0 <= score['confidence'] <= 1.0

    def test_claim_argument(self, signal_enricher):
        """Test scoring for claim-type arguments."""
        text = "La política social es fundamental para reducir la pobreza"
        score = signal_enricher.score_argument_with_signals(text, "claim", "PA02")

        assert 'final_score' in score
        assert 'confidence' in score


class TestTemporalMarkerExtraction:
    """Test suite for temporal marker extraction."""

    def test_year_extraction(self, signal_enricher):
        """Test extraction of year markers."""
        text = "El plan se ejecutará en 2024, 2025 y 2026"
        markers = signal_enricher.extract_temporal_markers_with_signals(text, "PA01")

        year_markers = [m for m in markers if m['type'] == 'YEAR']
        assert len(year_markers) > 0

    def test_date_extraction(self, signal_enricher):
        """Test extraction of date markers."""
        text = "La vigencia inicia el 01/01/2024 y termina el 31/12/2025"
        markers = signal_enricher.extract_temporal_markers_with_signals(text, "PA01")

        date_markers = [m for m in markers if m['type'] in ['DATE', 'YEAR', 'PERIOD']]
        assert len(date_markers) > 0

    def test_horizon_extraction(self, signal_enricher):
        """Test extraction of temporal horizon markers."""
        text = "Se implementarán medidas de corto plazo, mediano plazo y largo plazo"
        markers = signal_enricher.extract_temporal_markers_with_signals(text, "PA01")

        horizon_markers = [m for m in markers if m['type'] == 'HORIZON']
        assert len(horizon_markers) > 0


class TestFactoryFunction:
    """Test suite for factory function."""

    def test_create_signal_enricher_without_path(self):
        """Test factory function without questionnaire path."""
        enricher = create_signal_enricher()
        assert enricher is not None
        assert isinstance(enricher, SignalEnricher)

    def test_create_signal_enricher_with_path(self):
        """
        Test factory function with questionnaire path.
            This test checks that create_signal_enricher handles a non-existent file
```

```
gracefully.
        """
        path = Path("/tmp/test_questionnaire.json")
        # Ensure the file does not exist
        if path.exists():
            path.unlink()
        enricher = create_signal_enricher(questionnaire_path=path)
        assert enricher is not None
        assert isinstance(enricher, SignalEnricher)
        # Enricher should not be initialized when file doesn't exist
        assert not enricher._initialized


@pytest.mark.integration
class TestSignalEnrichmentIntegration:
    """Integration tests for signal enrichment with Phase 1."""

    def test_end_to_end_enrichment_flow(self, signal_enricher, sample_chunks):
        """Test complete enrichment flow from entity to metrics."""
        # 1. Enrich entities
        entity_enrichment = signal_enricher.enrich_entity_with_signals(
            "Ministerio de Hacienda", "ACTOR", "PA01"
        )
        assert entity_enrichment['signal_importance'] > 0

        # 2. Extract causal markers
        text = "La política causa efectos positivos"
             causal_markers  =  signal_enricher.extract_causal_markers_with_signals(text,
"PA01")
        assert len(causal_markers) > 0

        # 3. Score arguments
        arg_score = signal_enricher.score_argument_with_signals(
            "Las cifras muestran mejoras", "evidence", "PA01"
        )
        assert arg_score['final_score'] > 0

        # 4. Compute coverage metrics
        metrics = signal_enricher.compute_signal_coverage_metrics(sample_chunks)
        assert metrics['total_chunks'] == len(sample_chunks)

        # 5. Get provenance report
        report = signal_enricher.get_provenance_report()
        assert 'initialized' in report


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

tests/test_phase1_type_integration.py

```python
#!/usr/bin/env python3
"""
Phase 1 Type Integration Test
==============================

Tests the correct insertion of PolicyArea and DimensionCausal enum types
in Phase 1 and their proper propagation through the CPP production cycle.

This validates the requirement:
"Verifica la correcta inserción de types en fase 1 y garantiza su agregación
de valor en el ciclo de subfases de producción del CPP."
"""

import sys
from pathlib import Path

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


def test_canonical_types_import():
    """Test that canonical types can be imported"""
    print("\n" + "="*80)
    print("TEST 1: Canonical Types Import")
    print("="*80)

    try:
        from farfan_pipeline.core.types import PolicyArea, DimensionCausal
        print("  ? PolicyArea enum imported successfully")
        print("  ? DimensionCausal enum imported successfully")

        # Verify enum values
        assert hasattr(PolicyArea, 'PA01'), "PolicyArea missing PA01"
        assert hasattr(PolicyArea, 'PA10'), "PolicyArea missing PA10"
        print(f"  ? PolicyArea has PA01-PA10 values")

        assert hasattr(DimensionCausal, 'DIM01_INSUMOS'), "DimensionCausal missing
DIM01"
        assert hasattr(DimensionCausal, 'DIM06_CAUSALIDAD'), "DimensionCausal missing
DIM06"
        print(f"  ? DimensionCausal has DIM01-DIM06 values")

        print("\n  ? PASSED: Canonical types import successfully")
        return True
    except ImportError as e:
        print(f"  ? FAILED: Cannot import canonical types: {e}")
        return False


def test_phase1_models_have_enum_fields():
    """Test that Phase 1 models have enum fields"""
    print("\n" + "="*80)
    print("TEST 2: Phase 1 Models Have Enum Fields")
    print("="*80)

    try:
        # Import phase1_models using standard import
        from canonic_phases.Phase_one import phase1_models
        Chunk = phase1_models.Chunk
        SmartChunk = phase1_models.SmartChunk
        CANONICAL_TYPES_AVAILABLE = phase1_models.CANONICAL_TYPES_AVAILABLE

        print(f"  ? Chunk imported successfully")
        print(f"  ? SmartChunk imported successfully")
        print(f"  ? CANONICAL_TYPES_AVAILABLE = {CANONICAL_TYPES_AVAILABLE}")

        # Check Chunk has enum fields
        chunk = Chunk()
        assert hasattr(chunk, 'policy_area'), "Chunk missing policy_area field"
        assert hasattr(chunk, 'dimension'), "Chunk missing dimension field"
        print("  ? Chunk has policy_area and dimension fields")

        # Check SmartChunk has enum fields (via dataclass fields)
        from dataclasses import fields
        smart_chunk_fields = {f.name for f in fields(SmartChunk)}
        assert 'policy_area' in smart_chunk_fields, "SmartChunk missing policy_area field"
        assert 'dimension' in smart_chunk_fields, "SmartChunk missing dimension field"
        print("  ? SmartChunk has policy_area and dimension fields")

        print("\n  ? PASSED: Phase 1 models have enum fields")
        return True
    except Exception as e:
        print(f"  ? FAILED: {e}")
        import traceback
        traceback.print_exc()
        return False


def test_smartchunk_enum_conversion():
    """Test that SmartChunk automatically converts string IDs to enums"""
    print("\n" + "="*80)
    print("TEST 3: SmartChunk Enum Conversion")
    print("="*80)

    try:
        # Import directly from modules to avoid __init__.py dependency issues
        import sys
        import importlib.util

        spec = importlib.util.spec_from_file_location(
            "phase1_models",
            Path(__file__).parent.parent / "src/canonic_phases/Phase_one/phase1_models.py"
        )
        phase1_models = importlib.util.module_from_spec(spec)
        sys.modules['phase1_models_test2'] = phase1_models
        spec.loader.exec_module(phase1_models)

        SmartChunk = phase1_models.SmartChunk
        CANONICAL_TYPES_AVAILABLE = phase1_models.CANONICAL_TYPES_AVAILABLE

        if not CANONICAL_TYPES_AVAILABLE:
                  print("   ? SKIPPED: Canonical types not available, enum conversion
disabled")
            return True

        from farfan_pipeline.core.types import PolicyArea, DimensionCausal

        # Create SmartChunk with PA01-DIM01
        sc = SmartChunk(chunk_id="PA01-DIM01", text="Test chunk")

        # Verify string IDs are auto-derived
        assert sc.policy_area_id == "PA01", f"Expected PA01, got {sc.policy_area_id}"
        assert sc.dimension_id == "DIM01", f"Expected DIM01, got {sc.dimension_id}"
        print("  ? String IDs auto-derived correctly")

        # Verify enum types are set
        assert sc.policy_area is not None, "policy_area enum is None"
        assert sc.dimension is not None, "dimension enum is None"
        print("  ? Enum types are not None")

        # Verify correct enum values
               assert  sc.policy_area  ==  PolicyArea.PA01,  f"Expected  PA01  enum,  got
{sc.policy_area}"
         assert sc.dimension == DimensionCausal.DIM01_INSUMOS, f"Expected DIM01_INSUMOS,
got {sc.dimension}"
        print("  ? Enum values are correct (PA01, DIM01_INSUMOS)")

        # Test different PA×DIM combination
        sc2 = SmartChunk(chunk_id="PA10-DIM06", text="Test chunk 2")
        assert sc2.policy_area == PolicyArea.PA10, "PA10 enum not set correctly"
          assert sc2.dimension == DimensionCausal.DIM06_CAUSALIDAD, "DIM06 enum not set
correctly"
        print("  ? Multiple PA×DIM combinations work (PA10-DIM06)")

        print("\n  ? PASSED: SmartChunk enum conversion works correctly")
        return True
    except Exception as e:
        print(f"  ? FAILED: {e}")
        import traceback
        traceback.print_exc()
        return False


def test_cpp_models_have_enum_fields():
    """Test that CPP models have enum fields"""
    print("\n" + "="*80)
    print("TEST 4: CPP Models Have Enum Fields")
    print("="*80)
    try:
        # Import directly from modules to avoid __init__.py dependency issues
        import sys
        import importlib.util

        spec = importlib.util.spec_from_file_location(
            "cpp_models",
            Path(__file__).parent.parent / "src/canonic_phases/Phase_one/cpp_models.py"
        )
        cpp_models = importlib.util.module_from_spec(spec)
        sys.modules['cpp_models_test'] = cpp_models
        spec.loader.exec_module(cpp_models)

        LegacyChunk = cpp_models.LegacyChunk
        CANONICAL_TYPES_AVAILABLE = cpp_models.CANONICAL_TYPES_AVAILABLE

        print(f"  ? LegacyChunk imported successfully")
        print(f"  ? CANONICAL_TYPES_AVAILABLE = {CANONICAL_TYPES_AVAILABLE}")

        # Check LegacyChunk has enum fields
        from dataclasses import fields
        legacy_fields = {f.name for f in fields(LegacyChunk)}
        assert 'policy_area' in legacy_fields, "LegacyChunk missing policy_area field"
        assert 'dimension' in legacy_fields, "LegacyChunk missing dimension field"
        print("  ? LegacyChunk has policy_area and dimension fields")

        print("\n  ? PASSED: CPP models have enum fields")
        return True
    except Exception as e:
        print(f"  ? FAILED: {e}")
        import traceback
        traceback.print_exc()
        return False


def test_enum_value_aggregation():
    """Test that enum types enable value aggregation"""
    print("\n" + "="*80)
    print("TEST 5: Enum Value Aggregation")
    print("="*80)

    try:
        # Import directly from modules to avoid __init__.py dependency issues
        import sys
        import importlib.util

        spec = importlib.util.spec_from_file_location(
            "phase1_models",
                                                          Path(__file__).parent.parent     /
"src/canonic_phases/Phase_one/phase1_models.py"
        )
        phase1_models = importlib.util.module_from_spec(spec)
        sys.modules['phase1_models_test3'] = phase1_models
        spec.loader.exec_module(phase1_models)

        SmartChunk = phase1_models.SmartChunk
        CANONICAL_TYPES_AVAILABLE = phase1_models.CANONICAL_TYPES_AVAILABLE

        if not CANONICAL_TYPES_AVAILABLE:
                print("  ? SKIPPED: Canonical types not available, aggregation test
skipped")
            return True

        from farfan_pipeline.core.types import PolicyArea, DimensionCausal

        # Create multiple chunks
        chunks = [
            SmartChunk(chunk_id=f"{pa}-{dim}", text=f"Chunk {pa}-{dim}")
            for pa in ["PA01", "PA02", "PA03"]
            for dim in ["DIM01", "DIM02"]
        ]

        print(f"  ? Created {len(chunks)} test chunks")

        # Test aggregation by PolicyArea
        pa01_chunks = [c for c in chunks if c.policy_area == PolicyArea.PA01]
        assert len(pa01_chunks) == 2, f"Expected 2 PA01 chunks, got {len(pa01_chunks)}"
        print(f"  ? Aggregation by PolicyArea.PA01: {len(pa01_chunks)} chunks")

        # Test aggregation by DimensionCausal
                    dim01_chunks = [c for c in chunks if c.dimension ==
DimensionCausal.DIM01_INSUMOS]
                assert len(dim01_chunks) == 3, f"Expected 3 DIM01 chunks, got
{len(dim01_chunks)}"
          print(f"  ? Aggregation by DimensionCausal.DIM01_INSUMOS: {len(dim01_chunks)}
chunks")

        # Test combined aggregation
        pa02_dim02_chunks = [c for c in chunks
                             if c.policy_area == PolicyArea.PA02
                             and c.dimension == DimensionCausal.DIM02_ACTIVIDADES]
             assert len(pa02_dim02_chunks) == 1, f"Expected 1 PA02-DIM02 chunk, got
{len(pa02_dim02_chunks)}"
            print(f"  ? Combined aggregation (PA02 + DIM02): {len(pa02_dim02_chunks)}
chunk")

        # Verify enum comparison is faster than string comparison
        import time
        iterations = 10000

        # String comparison
        string_total = 0
        start = time.perf_counter()
        for _ in range(iterations):
            result = [c for c in chunks if c.policy_area_id == "PA01"]
            string_total += len(result)
        string_time = time.perf_counter() - start
        # Enum comparison
        enum_total = 0
        start = time.perf_counter()
        for _ in range(iterations):
            result = [c for c in chunks if c.policy_area == PolicyArea.PA01]
            enum_total += len(result)
        enum_time = time.perf_counter() - start

             print(f"   ?  Performance:  String={string_time:.4f}s,  Enum={enum_time:.4f}s
(totals: string={string_total}, enum={enum_total})")
            print(f"   ?  Enum  comparison  is  {'faster'  if  enum_time  <  string_time  else
'comparable'}")

        print("\n  ? PASSED: Enum value aggregation works correctly")
        return True
    except Exception as e:
        print(f"  ? FAILED: {e}")
        import traceback
        traceback.print_exc()
        return False



def test_type_propagation_metadata():
    """Test that type propagation metadata is tracked"""
    print("\n" + "="*80)
    print("TEST 6: Type Propagation Metadata")
    print("="*80)

    try:
        # This test verifies that the CPP metadata includes type propagation info
        # We can't run full Phase 1 execution here, so we verify the structure

        expected_metadata_keys = [
            'chunks_with_enums',
            'coverage_percentage',
            'canonical_types_available',
            'enum_ready_for_aggregation'
        ]

        print(f"  ? Expected metadata keys defined:")
        for key in expected_metadata_keys:
            print(f"    - {key}")

        # Check the source code directly
                                  phase1_file     =     Path(__file__).parent.parent    /
"src/canonic_phases/Phase_one/phase1_spc_ingestion_full.py"
        source = phase1_file.read_text()

        # Check if type propagation code exists
         assert 'type_propagation' in source, "type_propagation metadata not found in CPP
construction"
             assert  'chunks_with_enums'  in  source,  "chunks_with_enums  not  tracked  in
metadata"
```

```
            assert 'enum_ready_for_aggregation' in source, "enum_ready_for_aggregation not
tracked"
        print("  ? Type propagation metadata is tracked in CPP construction")

        print("\n  ? PASSED: Type propagation metadata structure verified")
        return True
    except Exception as e:
        print(f"  ? FAILED: {e}")
        import traceback
        traceback.print_exc()
        return False


def main():
    """Run all tests"""
    print("\n" + "="*80)
    print("  PHASE 1 TYPE INTEGRATION TEST SUITE")
    print("  Verifying PolicyArea and DimensionCausal enum integration")
    print("="*80)

    tests = [
        test_canonical_types_import,
        test_phase1_models_have_enum_fields,
        test_smartchunk_enum_conversion,
        test_cpp_models_have_enum_fields,
        test_enum_value_aggregation,
        test_type_propagation_metadata,
    ]

    results = []
    for test_func in tests:
        try:
            result = test_func()
            results.append(result)
        except Exception as e:
            print(f"\n  ? EXCEPTION in {test_func.__name__}: {e}")
            import traceback
            traceback.print_exc()
            results.append(False)

    # Summary
    print("\n" + "="*80)
    print("  TEST SUMMARY")
    print("="*80)
    passed = sum(results)
    total = len(results)
    print(f"  Passed: {passed}/{total}")
    print(f"  Failed: {total - passed}/{total}")

    if all(results):
        print("\n  ? ALL TESTS PASSED")
        print("  Type integration verified successfully!")
        return 0
    else:
        print("\n  ? SOME TESTS FAILED")
        print("  Type integration needs attention")
        return 1


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