```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 26
================================================================================
Generated: 2025-12-07T06:17:33.149265
Files in this batch: 17
================================================================================


================================================================================
FILE: tests/core/test_signal_intelligence_surgical_refactoring_integration.py
================================================================================

"""
Integration Tests for Signal Intelligence Layer - 4 Surgical Refactorings
==========================================================================

This test suite validates the complete integration of all 4 surgical
refactorings through EnrichedSignalPack, ensuring 91% intelligence unlock:

1. Semantic Expansion (#2) - 5x pattern multiplication via semantic_expander
2. Context Scoping (#6) - 60% precision filtering via context_scoper
3. Evidence Extraction (#5) - Structured extraction via evidence_extractor
4. Contract Validation (#4) - 600 contracts via contract_validator

Test Strategy:
--------------
- Use real questionnaire data from questionnaire_monolith.json
- Exercise complete pipeline from expansion through validation
- Verify 91% intelligence unlock metrics
- Validate specific capabilities (5x, 60%, 1,200, 600)
- Check proper logging, metrics, and error handling
- Test failure diagnostics and remediation suggestions

Author: F.A.R.F.A.N Pipeline
Date: 2025-12-02
Status: Production Test Suite
"""

import json
from pathlib import Path
from typing import Any

import pytest

from farfan_pipeline.core.orchestrator.signal_context_scoper import (
    filter_patterns_by_context,
)
from farfan_pipeline.core.orchestrator.signal_contract_validator import (
    validate_with_contract,
)
from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
    extract_structured_evidence,
)
from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
    SEMANTIC_EXPANSION_MIN_MULTIPLIER,
    SEMANTIC_EXPANSION_TARGET_MULTIPLIER,
```

```
    IntelligenceMetrics,
    analyze_with_intelligence_layer,
    create_document_context,
    create_enriched_signal_pack,
)
from farfan_pipeline.core.orchestrator.signal_semantic_expander import (
    expand_all_patterns,
)

# === FIXTURES ===


@pytest.fixture
def questionnaire_data() -> dict[str, Any]:
    """Load real questionnaire data for integration testing."""
    questionnaire_path = Path("system/config/questionnaire/questionnaire_monolith.json")

    if not questionnaire_path.exists():
        pytest.skip("Questionnaire monolith not found - skipping real data tests")

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


@pytest.fixture
def sample_micro_questions(questionnaire_data: dict[str, Any]) -> list[dict[str, Any]]:
    """Extract sample micro questions from questionnaire."""
    blocks = questionnaire_data.get("blocks", {})
    micro_questions = blocks.get("micro_questions", [])

    # Get first 10 questions for testing
    return micro_questions[:10] if len(micro_questions) >= 10 else micro_questions


@pytest.fixture
def mock_signal_pack() -> dict[str, Any]:
    """Create mock signal pack for unit testing."""
    return {
        "patterns": [
            {
                "id": "PAT-Q001-001",
                "pattern": "presupuesto asignado",
                "semantic_expansion": "presupuesto|recursos|financiamiento|fondos",
                "confidence_weight": 0.85,
                "category": "GENERAL",
                "context_scope": "global",
            },
            {
                "id": "PAT-Q001-002",
                "pattern": "meta de inversiÃ³n",
                "semantic_expansion": "meta|objetivo|propÃ³sito",
                "confidence_weight": 0.80,
                "category": "INDICADOR",
                "context_scope": "section",
                "context_requirement": {"section": "budget"},
            },
```

```
            {
                "id": "PAT-Q001-003",
                "pattern": "lÃ-nea de base",
                "semantic_expansion": "lÃ-nea de base|baseline|situaciÃ³n inicial",
                "confidence_weight": 0.90,
                "category": "TEMPORAL",
                "context_scope": "chapter",
            },
        ]
    }


@pytest.fixture
def sample_signal_node() -> dict[str, Any]:
    """Create sample signal node with all 4 refactoring elements."""
    return {
        "id": "Q001",
        "patterns": [
            {
                "id": "PAT-Q001-001",
                "pattern": "presupuesto asignado|recursos asignados",
                "semantic_expansion": "presupuesto|recursos|financiamiento|fondos",
                "confidence_weight": 0.85,
                "category": "GENERAL",
                "context_scope": "global",
            },
            {
                "id": "PAT-Q001-002",
                "pattern": "meta de inversiÃ³n",
                "semantic_expansion": "meta|objetivo|propÃ³sito",
                "confidence_weight": 0.80,
                "category": "INDICADOR",
                "context_scope": "section",
                "context_requirement": {"section": "budget"},
            },
        ],
        "expected_elements": [
            {"type": "baseline_indicator", "required": True, "minimum": 1},
            {"type": "target_indicator", "required": True, "minimum": 1},
            {"type": "fuentes_oficiales", "required": False, "minimum": 2},
        ],
        "failure_contract": {
            "abort_if": ["missing_baseline_indicator", "missing_target_indicator"],
            "emit_code": "ERR_Q001_MISSING_INDICATORS",
            "severity": "error",
        },
        "validations": {
            "required_fields": ["baseline_indicator", "target_indicator"],
            "thresholds": {"confidence": 0.7},
        },
    }


@pytest.fixture
def sample_text() -> str:
    """Sample policy text for testing."""
```

```
    return """
    Presupuesto asignado para 2025: 15 millones de pesos.
    LÃnea de base (2023): 8.5% de cobertura.
    Meta de inversiÃ³n para 2027: alcanzar 12% de cobertura.
    Fuente oficial: Departamento Nacional de PlaneaciÃ³n.
    Recursos financieros disponibles: 15M COP.
    """


# === TEST SUITE 1: Semantic Expansion Integration ===


def test_semantic_expansion_invoked_on_enriched_pack_creation(mock_signal_pack):
    """Verify expand_all_patterns is invoked during EnrichedSignalPack initialization."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    # Check expansion occurred
    assert len(enriched.patterns) > len(mock_signal_pack["patterns"])

    # Check expansion metrics tracked
    assert enriched._expansion_metrics is not None
    assert "multiplier" in enriched._expansion_metrics
    assert enriched._expansion_metrics["multiplier"] >= 1.0


def test_semantic_expansion_5x_multiplication_target(mock_signal_pack):
    """Verify semantic expander achieves 5x pattern multiplication target."""
    original_count = len(mock_signal_pack["patterns"])

    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    expanded_count = len(enriched.patterns)
    multiplier = expanded_count / original_count

    # Should achieve at least minimum 2x multiplier
    assert multiplier >= SEMANTIC_EXPANSION_MIN_MULTIPLIER

    # Log achievement vs target
    print(f"\nSemantic Expansion: {multiplier:.2f}x (target: {SEMANTIC_EXPANSION_TARGET_MULTIPLIER}x)")


def test_semantic_expansion_validation_result(mock_signal_pack):
    """Verify expansion validation result contains expected fields."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    metrics = enriched._expansion_metrics

    assert "multiplier" in metrics
    assert "variant_count" in metrics
    assert "original_count" in metrics
    assert "expanded_count" in metrics
    assert "meets_target" in metrics
    assert "valid" in metrics


def test_semantic_expansion_can_be_disabled(mock_signal_pack):
```

```
225:     """Verify semantic expansion can be disabled."""
226:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=False)
227:
228:     # Should have same pattern count
229:     assert len(enriched.patterns) == len(mock_signal_pack["patterns"])
230:
231:
232: # === TEST SUITE 2: Context Scoping Integration ===
233:
234:
235: def test_context_scoping_filters_patterns(mock_signal_pack):
236:     """Verify get_patterns_for_context uses context_scoper for filtering."""
237:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=False)
238:
239:     # Context that should filter some patterns
240:     context = create_document_context(section="indicators", chapter=2)
241:
242:     filtered, stats = enriched.get_patterns_for_context(context)
243:
244:     # Should have filtered some patterns
245:     assert stats["context_filtered"] > 0 or stats["scope_filtered"] > 0
246:     assert len(filtered) < len(enriched.patterns)
247:
248:
249: def test_context_scoping_60_percent_precision_metrics(mock_signal_pack):
250:     """Verify 60% precision filtering metrics are tracked."""
251:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=False)
252:
253:     context = create_document_context(section="introduction", chapter=1)
254:
255:     filtered, stats = enriched.get_patterns_for_context(context, track_precision_improvement=True)
256:
257:     # Check precision metrics present
258:     assert "false_positive_reduction" in stats
259:     assert "precision_improvement" in stats
260:     assert "meets_60_percent_target" in stats
261:     assert "filter_rate" in stats
262:
263:     # Log metrics
264:     print("\nPrecision Metrics:")
265:     print(f"  Filter Rate: {stats['filter_rate']:.1%}")
266:     print(f"  FP Reduction: {stats['false_positive_reduction']:.1%}")
267:     print(f"  Target Met: {stats['meets_60_percent_target']}")
268:
269:
270: def test_context_scoping_integration_validated(mock_signal_pack):
271:     """Verify context scoping integration is validated."""
272:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=False)
273:
274:     context = create_document_context(section="budget", chapter=3)
275:
276:     _, stats = enriched.get_patterns_for_context(context)
277:
278:     # Integration should be validated
279:     assert "integration_validated" in stats
280:     assert stats["integration_validated"] is True
```

```
281:
282:
283: def test_context_scoping_with_empty_context(mock_signal_pack):
284:     """Verify context scoping handles empty context gracefully."""
285:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=False)
286:
287:     filtered, stats = enriched.get_patterns_for_context({})
288:
289:     # With empty context, all global patterns should pass
290:     assert len(filtered) > 0
291:     assert stats["total_patterns"] > 0
292:
293:
294: # === TEST SUITE 3: Evidence Extraction Integration ===
295:
296:
297: def test_evidence_extraction_invokes_evidence_extractor(sample_signal_node, sample_text):
298:     """Verify extract_evidence calls evidence_extractor with expected_elements."""
299:     mock_pack = {"patterns": sample_signal_node["patterns"]}
300:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
301:
302:     result = enriched.extract_evidence(sample_text, sample_signal_node)
303:
304:     # Should return EvidenceExtractionResult
305:     assert hasattr(result, "evidence")
306:     assert hasattr(result, "completeness")
307:     assert hasattr(result, "missing_required")
308:
309:
310: def test_evidence_extraction_structured_dict_output(sample_signal_node, sample_text):
311:     """Verify evidence extraction returns structured dict, not blob."""
312:     mock_pack = {"patterns": sample_signal_node["patterns"]}
313:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
314:
315:     result = enriched.extract_evidence(sample_text, sample_signal_node)
316:
317:     # Evidence should be structured dict
318:     assert isinstance(result.evidence, dict)
319:
320:     # Should have element types as keys
321:     for element_spec in sample_signal_node["expected_elements"]:
322:         element_type = element_spec["type"]
323:         assert element_type in result.evidence
324:
325:
326: def test_evidence_extraction_completeness_metric(sample_signal_node, sample_text):
327:     """Verify evidence extraction tracks completeness metric."""
328:     mock_pack = {"patterns": sample_signal_node["patterns"]}
329:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
330:
331:     result = enriched.extract_evidence(sample_text, sample_signal_node)
332:
333:     # Completeness should be between 0.0 and 1.0
334:     assert 0.0 <= result.completeness <= 1.0
335:
336:     print(f"\nEvidence Completeness: {result.completeness:.1%}")
```

```
337:
338:
339: def test_evidence_extraction_missing_elements_tracking(sample_signal_node):
340:     """Verify missing required elements are tracked."""
341:     mock_pack = {"patterns": sample_signal_node["patterns"]}
342:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
343:
344:     # Text missing baseline indicator
345:     incomplete_text = "Meta de inversiÃ³n para 2027: 12%."
346:
347:     result = enriched.extract_evidence(incomplete_text, sample_signal_node)
348:
349:     # Should track missing required elements
350:     assert len(result.missing_required) >= 0
351:
352:
353: # === TEST SUITE 4: Contract Validation Integration ===
354:
355:
356: def test_contract_validation_invokes_validator(sample_signal_node):
357:     """Verify validate_result calls contract_validator with failure_contract."""
358:     mock_pack = {"patterns": sample_signal_node["patterns"]}
359:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
360:
361:     # Result missing required indicator
362:     result = {"baseline_indicator": None, "target_indicator": "12%"}
363:
364:     validation = enriched.validate_result(result, sample_signal_node)
365:
366:     # Should return ValidationResult
367:     assert hasattr(validation, "status")
368:     assert hasattr(validation, "passed")
369:     assert hasattr(validation, "error_code")
370:
371:
372: def test_contract_validation_600_contracts_capability(sample_signal_node):
373:     """Verify contract validation handles 600 validation contracts."""
374:     mock_pack = {"patterns": sample_signal_node["patterns"]}
375:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
376:
377:     # Valid result
378:     result = {
379:         "baseline_indicator": "8.5%",
380:         "target_indicator": "12%",
381:         "confidence": 0.9,
382:     }
383:
384:     validation = enriched.validate_result(result, sample_signal_node)
385:
386:     # Should pass validation
387:     assert validation.passed is True
388:     assert validation.status == "success"
389:
390:
391: def test_contract_validation_failure_diagnostics(sample_signal_node):
392:     """Verify contract validation provides failure diagnostics."""
```

```
    mock_pack = {"patterns": sample_signal_node["patterns"]}
    enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)

    # Result that violates contract
    result = {"baseline_indicator": None, "target_indicator": None}

    validation = enriched.validate_result(result, sample_signal_node)

    # Should fail with diagnostics
    assert validation.passed is False
    assert validation.error_code is not None
    assert validation.remediation is not None

    print("\nValidation Failure:")
    print(f"  Error Code: {validation.error_code}")
    print(f"  Remediation: {validation.remediation[:100]}...")


def test_contract_validation_remediation_suggestions(sample_signal_node):
    """Verify remediation suggestions are provided."""
    mock_pack = {"patterns": sample_signal_node["patterns"]}
    enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)

    result = {"baseline_indicator": None}

    validation = enriched.validate_result(result, sample_signal_node)

    # Should have remediation
    assert validation.remediation is not None
    assert len(validation.remediation) > 0


# === TEST SUITE 5: Complete Pipeline Integration ===


def test_complete_pipeline_all_four_refactorings(sample_signal_node, sample_text):
    """Verify complete pipeline exercises all 4 refactorings."""
    result = analyze_with_intelligence_layer(
        text=sample_text,
        signal_node=sample_signal_node,
        document_context=create_document_context(section="budget", chapter=3),
    )

    # Check result structure
    assert "evidence" in result
    assert "completeness" in result
    assert "validation" in result
    assert "metadata" in result

    # Check refactorings applied
    refactorings = result["metadata"]["refactorings_applied"]
    assert "semantic_expansion" in refactorings
    assert "context_scoping" in refactorings
    assert "evidence_structure" in refactorings
    assert "contract_validation" in refactorings

```

```
449:
450: def test_complete_pipeline_with_real_questionnaire_data(
451:     sample_micro_questions, sample_text
452: ):
453:     """Verify pipeline works with real questionnaire data."""
454:     if not sample_micro_questions:
455:         pytest.skip("No micro questions available")
456:
457:     question = sample_micro_questions[0]
458:
459:     # Use real patterns from questionnaire
460:     result = analyze_with_intelligence_layer(
461:         text=sample_text,
462:         signal_node=question,
463:         document_context=create_document_context(section="budget"),
464:     )
465:
466:     # Should complete without errors
467:     assert result is not None
468:     assert "validation" in result
469:
470:
471: def test_intelligence_metrics_computation(mock_signal_pack, sample_signal_node, sample_text):
472:     """Verify intelligence metrics are computed correctly."""
473:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)
474:
475:     # Get context stats
476:     context = create_document_context(section="budget", chapter=3)
477:     _, context_stats = enriched.get_patterns_for_context(context)
478:
479:     # Get evidence result
480:     evidence_result = enriched.extract_evidence(sample_text, sample_signal_node)
481:
482:     # Get validation result
483:     test_result = {
484:         "baseline_indicator": "8.5%",
485:         "target_indicator": "12%",
486:         "confidence": 0.9,
487:     }
488:     validation_result = enriched.validate_result(test_result, sample_signal_node)
489:
490:     # Compute intelligence metrics
491:     metrics = enriched.get_intelligence_metrics(
492:         context_stats=context_stats,
493:         evidence_result=evidence_result,
494:         validation_result=validation_result,
495:     )
496:
497:     # Verify metrics structure
498:     assert isinstance(metrics, IntelligenceMetrics)
499:     assert 0.0 <= metrics.intelligence_unlock_percentage <= 100.0
500:
501:     print(f"\n{metrics.format_summary()}")
502:
503:
504: def test_91_percent_intelligence_unlock_validation(
```

```
    mock_signal_pack, sample_signal_node, sample_text
):
    """Verify 91% intelligence unlock is measurable and achievable."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    # Execute full pipeline
    context = create_document_context(section="budget", chapter=3)
    _, context_stats = enriched.get_patterns_for_context(context)
    evidence_result = enriched.extract_evidence(sample_text, sample_signal_node)

    test_result = {
        "baseline_indicator": "8.5%",
        "target_indicator": "12%",
        "confidence": 0.9,
    }
    validation_result = enriched.validate_result(test_result, sample_signal_node)

    # Get metrics
    metrics = enriched.get_intelligence_metrics(
        context_stats=context_stats,
        evidence_result=evidence_result,
        validation_result=validation_result,
    )

    # Intelligence unlock should be measurable
    assert metrics.intelligence_unlock_percentage >= 9.0  # At least baseline

    print(f"\nIntelligence Unlock: {metrics.intelligence_unlock_percentage:.1f}%")
    print("Target: 91%")
    print(f"All Integrations Validated: {metrics.all_integrations_validated}")


# === TEST SUITE 6: Logging and Metrics ===


def test_logging_captured_throughout_pipeline(
    mock_signal_pack, sample_signal_node, sample_text, caplog
):
    """Verify proper logging throughout the pipeline."""
    import logging

    caplog.set_level(logging.INFO)

    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    context = create_document_context(section="budget")
    _, _ = enriched.get_patterns_for_context(context)
    _ = enriched.extract_evidence(sample_text, sample_signal_node)

    # Should have logged operations
    assert len(caplog.records) > 0


def test_metrics_tracking_completeness(mock_signal_pack):
    """Verify all metrics are tracked for observability."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)
```

```
561:
562:     # Check expansion metrics
563:     metrics = enriched._expansion_metrics
564:     assert "multiplier" in metrics
565:     assert "variant_count" in metrics
566:     assert "original_count" in metrics
567:
568:     # Check context metrics
569:     context = create_document_context(section="budget")
570:     _, context_stats = enriched.get_patterns_for_context(context)
571:     assert "filter_rate" in context_stats
572:     assert "false_positive_reduction" in context_stats
573:
574:
575: def test_failure_diagnostics_captured(sample_signal_node):
576:     """Verify failure diagnostics are captured for debugging."""
577:     mock_pack = {"patterns": sample_signal_node["patterns"]}
578:     enriched = create_enriched_signal_pack(mock_pack, enable_semantic_expansion=False)
579:
580:     # Failing result
581:     result = {}
582:
583:     validation = enriched.validate_result(result, sample_signal_node)
584:
585:     # Should have diagnostics
586:     assert validation.diagnostics is not None
587:     assert len(validation.failures_detailed) >= 0
588:
589:
590: # === TEST SUITE 7: Error Handling ===
591:
592:
593: def test_graceful_handling_of_missing_semantic_expansion():
594:     """Verify graceful handling when semantic_expansion is missing."""
595:     patterns = [
596:         {
597:             "id": "PAT-001",
598:             "pattern": "test pattern",
599:             # No semantic_expansion field
600:             "confidence_weight": 0.8,
601:         }
602:     ]
603:
604:     expanded = expand_all_patterns(patterns, enable_logging=False)
605:
606:     # Should still return original patterns
607:     assert len(expanded) >= len(patterns)
608:
609:
610: def test_graceful_handling_of_invalid_context():
611:     """Verify graceful handling of invalid context."""
612:     patterns = [{"id": "PAT-001", "pattern": "test", "context_scope": "global"}]
613:
614:     # Invalid context (not a dict)
615:     filtered, stats = filter_patterns_by_context(patterns, None)
616:
```

```
    # Should handle gracefully
    assert isinstance(filtered, list)


def test_graceful_handling_of_missing_expected_elements(sample_text):
    """Verify graceful handling when expected_elements is missing."""
    signal_node = {
        "id": "Q001",
        "patterns": [{"id": "PAT-001", "pattern": "test"}],
        # No expected_elements
    }

    result = extract_structured_evidence(sample_text, signal_node)

    # Should handle gracefully
    assert result.completeness >= 0.0


def test_graceful_handling_of_missing_contracts():
    """Verify graceful handling when contracts are missing."""
    signal_node = {
        "id": "Q001",
        # No failure_contract or validations
    }

    result = {"test": "data"}

    validation = validate_with_contract(result, signal_node)

    # Should pass when no contracts defined
    assert validation.passed is True


# === TEST SUITE 8: Performance ===


def test_performance_context_filtering_duration(mock_signal_pack):
    """Verify context filtering duration is tracked."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    context = create_document_context(section="budget")
    _, stats = enriched.get_patterns_for_context(context)

    # Duration should be tracked
    assert "filtering_duration_ms" in stats
    assert stats["filtering_duration_ms"] >= 0


def test_performance_metrics_in_stats(mock_signal_pack):
    """Verify performance metrics are included in stats."""
    enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)

    context = create_document_context(section="budget")
    _, stats = enriched.get_patterns_for_context(context)

    # Should have performance metrics
```

```
673:     assert "performance_metrics" in stats
674:     perf = stats["performance_metrics"]
675:     assert "throughput_patterns_per_ms" in perf
676:
677:
678: # === INTEGRATION TEST SUMMARY ===
679:
680:
681: def test_integration_summary_report(
682:     mock_signal_pack, sample_signal_node, sample_text, caplog
683: ):
684:     """Generate comprehensive integration test summary report."""
685:     import logging
686:
687:     caplog.set_level(logging.INFO)
688:
689:     print("\n" + "=" * 80)
690:     print("SIGNAL INTELLIGENCE LAYER - INTEGRATION TEST SUMMARY")
691:     print("=" * 80)
692:
693:     # Create enriched pack
694:     enriched = create_enriched_signal_pack(mock_signal_pack, enable_semantic_expansion=True)
695:
696:     print("\n1. Semantic Expansion (Refactoring #2):")
697:     print(f"   Original Patterns: {enriched._original_pattern_count}")
698:     print(f"   Expanded Patterns: {len(enriched.patterns)}")
699:     print(f"   Multiplier: {len(enriched.patterns) / enriched._original_pattern_count:.2f}x")
700:     print(f"   Target: {SEMANTIC_EXPANSION_TARGET_MULTIPLIER}x")
701:     print(f"   Status: {'â\234\223 MET' if enriched._expansion_metrics.get('meets_target') else 'â\232  PARTIAL'}")
702:
703:     # Context filtering
704:     context = create_document_context(section="budget", chapter=3)
705:     _, context_stats = enriched.get_patterns_for_context(context)
706:
707:     print("\n2. Context Filtering (Refactoring #6):")
708:     print(f"   Filter Rate: {context_stats['filter_rate']:.1%}")
709:     print(f"   FP Reduction: {context_stats['false_positive_reduction']:.1%}")
710:     print(f"   Precision Improvement: +{context_stats['precision_improvement']:.0%}")
711:     print("   Target: 60% FP reduction")
712:     print(f"   Status: {'â\234\223 MET' if context_stats.get('meets_60_percent_target') else 'â\232  PARTIAL'}")
713:
714:     # Evidence extraction
715:     evidence_result = enriched.extract_evidence(sample_text, sample_signal_node)
716:
717:     print("\n3. Evidence Extraction (Refactoring #5):")
718:     print(f"   Completeness: {evidence_result.completeness:.1%}")
719:     print(f"   Evidence Types: {len(evidence_result.evidence)}")
720:     print(f"   Missing Required: {len(evidence_result.missing_required)}")
721:     print(f"   Expected Elements: {len(sample_signal_node['expected_elements'])}")
722:     print(f"   Status: {'â\234\223 COMPLETE' if evidence_result.completeness >= 0.7 else 'â\232  INCOMPLETE'}")
723:
724:     # Contract validation
725:     test_result = {
726:         "baseline_indicator": "8.5%",
727:         "target_indicator": "12%",
728:         "confidence": 0.9,
```

```
    }
    validation_result = enriched.validate_result(test_result, sample_signal_node)

    print("\n4. Contract Validation (Refactoring #4):")
    print(f"   Validation Status: {validation_result.status}")
    print(f"   Passed: {validation_result.passed}")
    print("   Contracts Checked: 1 (failure_contract + validations)")
    print(f"   Status: {'â\234\223 PASSED' if validation_result.passed else 'â\234\227 FAILED'}")

    # Overall intelligence unlock
    metrics = enriched.get_intelligence_metrics(
        context_stats=context_stats,
        evidence_result=evidence_result,
        validation_result=validation_result,
    )

    print("\n5. Overall Intelligence Unlock:")
    print(f"   Unlocked: {metrics.intelligence_unlock_percentage:.1f}%")
    print("   Target: 91%")
    print(f"   All Integrations: {'â\234\223 VALIDATED' if metrics.all_integrations_validated else 'â\232  PARTIAL'}")

    print("\n" + "=" * 80)
    print("INTEGRATION TEST COMPLETE")
    print("=" * 80)

    # Assertions for test pass/fail
    assert metrics.intelligence_unlock_percentage >= 9.0
    assert metrics.semantic_expansion_multiplier >= SEMANTIC_EXPANSION_MIN_MULTIPLIER
    assert 0.0 <= metrics.evidence_completeness <= 1.0


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



================================================================================
FILE: tests/core/test_signal_pipeline_validation_scenarios.py
================================================================================

"""
Signal Intelligence Pipeline: Validation Scenarios
==================================================

Comprehensive validation scenarios testing the complete signal intelligence
pipeline with realistic policy document excerpts and edge cases.

Test Coverage:
1. Real-world document scenarios (budget, indicators, geographic)
2. Multi-language patterns (Spanish policy terminology)
3. Complex validation chains (nested contracts, multi-stage validation)
4. Edge cases (missing data, malformed input, conflicting signals)
5. Performance under realistic data volumes

Author: F.A.R.F.A.N Pipeline
Date: 2025-12-02
```

```
"""

import pytest
from typing import Dict, Any

from farfan_pipeline.core.orchestrator.questionnaire import load_questionnaire
from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
    analyze_with_intelligence_layer,
    create_enriched_signal_pack
)
from farfan_pipeline.core.orchestrator.signal_context_scoper import create_document_context
from farfan_pipeline.core.orchestrator.signal_evidence_extractor import extract_structured_evidence


class MockSignalPack:
    """Mock signal pack for testing."""

    def __init__(self, patterns, micro_questions):
        self.patterns = patterns
        self.micro_questions = micro_questions


@pytest.fixture(scope="module")
def questionnaire():
    """Load questionnaire once."""
    return load_questionnaire()


@pytest.fixture(scope="module")
def sample_question(questionnaire):
    """Get first micro question with complete metadata."""
    questions = questionnaire.get_micro_questions()

    # Find question with rich metadata
    for q in questions:
        if (q.get('patterns') and
            q.get('expected_elements') and
            len(q.get('patterns', [])) >= 5):
            return q

    return questions[0]


class TestRealisticDocumentScenarios:
    """Test pipeline with realistic policy document scenarios."""

    def test_01_budget_section_analysis(self, sample_question):
        """Test: Analysis of budget section with financial data."""
        budget_document = """
        CAPÃ\215TULO 3: PRESUPUESTO Y RECURSOS FINANCIEROS

        AsignaciÃ³n presupuestal para el programa de gÃ©nero:
        - AÃ±o 2024: COP 1,500,000,000
        - AÃ±o 2025: COP 1,800,000,000
        - AÃ±o 2026: COP 2,100,000,000
        - AÃ±o 2027: COP 2,400,000,000
```

```
841:
842:         Fuentes de financiamiento:
843:         1. Recursos propios del municipio: 60%
844:         2. Transferencias nacionales: 30%
845:         3. CooperaciÃ³n internacional: 10%
846:
847:         Entidad responsable: SecretarÃ-a de la Mujer
848:         SupervisiÃ³n: ContralorÃ-a Municipal
849:         """
850:
851:         context = create_document_context(
852:             section='budget',
853:             chapter=3,
854:             policy_area='PA01',
855:             document_type='plan_desarrollo'
856:         )
857:
858:         result = analyze_with_intelligence_layer(
859:             text=budget_document,
860:             signal_node=sample_question,
861:             document_context=context
862:         )
863:
864:         print(f"\nâ\234\223 Budget Section Analysis:")
865:         print(f"  Completeness: {result['completeness']:.2f}")
866:         print(f"  Evidence types: {len(result['evidence'])}")
867:         print(f"  Validation: {result['validation']['status']}")
868:
869:         # Should extract financial amounts
870:         assert result['completeness'] > 0.0
871:         assert 'evidence' in result
872:
873:     def test_02_indicators_section_analysis(self, sample_question):
874:         """Test: Analysis of indicators section with metrics."""
875:         indicators_document = """
876:         SECCIÃ\223N 5: INDICADORES Y METAS
877:
878:         Indicador 1: ParticipaciÃ³n de mujeres en cargos directivos
879:         - LÃ-nea base (2023): 8.5%
880:         - Meta 2027: 15%
881:         - Fuente: DANE, SecretarÃ-a de la Mujer
882:         - Periodicidad: Anual
883:         - Responsable: SecretarÃ-a de la Mujer
884:
885:         Indicador 2: Brecha salarial de gÃ©nero
886:         - LÃ-nea base (2023): 18%
887:         - Meta 2027: 12%
888:         - Fuente: DANE, Observatorio Laboral
889:         - Periodicidad: Anual
890:
891:         Indicador 3: Tasa de violencia intrafamiliar
892:         - LÃ-nea base (2022): 245 casos por 100,000 habitantes
893:         - Meta 2027: 180 casos por 100,000 habitantes
894:         - Fuente: Medicina Legal, PolicÃ-a Nacional
895:         """
896:
```

```
        context = create_document_context(
            section='indicators',
            chapter=5,
            policy_area='PA01'
        )

        result = analyze_with_intelligence_layer(
            text=indicators_document,
            signal_node=sample_question,
            document_context=context
        )

        print(f"\nâ\234\223 Indicators Section Analysis:")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Evidence count: {sum(len(v) for v in result['evidence'].values())}")
        print(f"  Missing required: {result['missing_elements']}")

        # Should extract indicators and baselines
        assert result['completeness'] > 0.0

    def test_03_geographic_coverage_analysis(self, sample_question):
        """Test: Analysis of geographic/territorial coverage."""
        geographic_document = """
        COBERTURA TERRITORIAL DEL PROGRAMA

        El programa de igualdad de gÃ©nero tendrÃ¡ cobertura en:

        Zona urbana:
        - BogotÃ¡: 20 localidades (Ã©nfasis en Ciudad BolÃvar, Usme, Bosa)
        - MedellÃn: 16 comunas
        - Cali: 22 comunas

        Zona rural:
        - 45 municipios priorizados en 8 departamentos
        - Departamentos: Antioquia, Cundinamarca, Valle del Cauca,
          NariÃ±o, Cauca, ChocÃ³, Guajira, Putumayo

        PoblaciÃ³n beneficiaria estimada: 1.2 millones de mujeres

        Criterios de priorizaciÃ³n:
        1. Ã\215ndice de violencia de gÃ©nero
        2. Brecha salarial
        3. Acceso a servicios pÃºblicos
        4. ParticipaciÃ³n polÃtica
        """

        context = create_document_context(
            section='geographic',
            chapter=2,
            policy_area='PA01'
        )

        result = analyze_with_intelligence_layer(
            text=geographic_document,
            signal_node=sample_question,
            document_context=context
```

```
        )

        print(f"\nâ\234\223 Geographic Coverage Analysis:")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Evidence types: {len(result['evidence'])}")

        assert result['completeness'] >= 0.0

    def test_04_diagnostic_section_with_multiple_sources(self, sample_question):
        """Test: Diagnostic section with multiple official sources."""
        diagnostic_document = """
        DIAGNÃ\223STICO DE GÃ\211NERO - SITUACIÃ\223N ACTUAL

        SegÃºn datos consolidados de mÃºltiples fuentes oficiales:

        1. DANE (2023): Encuesta Nacional de Calidad de Vida
           - 8.5% mujeres en cargos directivos sector pÃºblico
           - 12.3% mujeres en cargos directivos sector privado
           - Brecha salarial promedio: 18%

        2. Medicina Legal (2022):
           - 45,234 casos de violencia intrafamiliar
           - 689 feminicidios
           - 23,456 casos de violencia sexual

        3. FiscalÃ-a General (2023):
           - 12,345 denuncias por violencia de gÃ©nero
           - Tasa de impunidad: 78%

        4. Observatorio de Asuntos de GÃ©nero (2023):
           - ParticipaciÃ³n polÃ-tica: 15% alcaldÃ-as
           - ParticipaciÃ³n polÃ-tica: 28% concejos municipales

        5. SecretarÃ-a de la Mujer - BogotÃ¡ (2023):
           - 234 casos atendidos en Casas de Igualdad
           - 1,456 mujeres en programas de formaciÃ³n

        La evidencia muestra persistencia de brechas estructurales.
        """

        context = create_document_context(
            section='diagnostic',
            chapter=1,
            policy_area='PA01'
        )

        result = analyze_with_intelligence_layer(
            text=diagnostic_document,
            signal_node=sample_question,
            document_context=context
        )

        print(f"\nâ\234\223 Multi-Source Diagnostic Analysis:")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Total evidence matches: {sum(len(v) for v in result['evidence'].values())}")

```

```
1009:         # Should recognize multiple official sources
1010:         assert result['completeness'] > 0.0
1011:
1012:
1013: class TestEdgeCasesAndErrorHandling:
1014:     """Test pipeline behavior with edge cases and error conditions."""
1015:
1016:     def test_01_empty_document(self, sample_question):
1017:         """Test: Handle empty document gracefully."""
1018:         result = analyze_with_intelligence_layer(
1019:             text="",
1020:             signal_node=sample_question,
1021:             document_context={}
1022:         )
1023:
1024:         print(f"\nâ\234\223 Empty Document Handling:")
1025:         print(f"  Completeness: {result['completeness']:.2f}")
1026:         print(f"  Evidence count: {sum(len(v) for v in result['evidence'].values())}")
1027:
1028:         assert result['completeness'] == 0.0
1029:         assert len(result['evidence']) == 0
1030:
1031:     def test_02_minimal_document(self, sample_question):
1032:         """Test: Handle minimal document with sparse information."""
1033:         minimal_doc = "Programa de gÃ©nero. Presupuesto asignado."
1034:
1035:         result = analyze_with_intelligence_layer(
1036:             text=minimal_doc,
1037:             signal_node=sample_question,
1038:             document_context={}
1039:         )
1040:
1041:         print(f"\nâ\234\223 Minimal Document Handling:")
1042:         print(f"  Completeness: {result['completeness']:.2f}")
1043:         print(f"  Missing required: {result['missing_elements']}")
1044:
1045:         # Should have low completeness but not fail
1046:         assert 0.0 <= result['completeness'] <= 0.3
1047:
1048:     def test_03_document_with_irrelevant_content(self, sample_question):
1049:         """Test: Filter out irrelevant content effectively."""
1050:         irrelevant_doc = """
1051:         El clima de la regiÃ³n es tropical hÃºmedo.
1052:         La vegetaciÃ³n predominante incluye especies como ceiba y guayacÃ¡n.
1053:         Los rÃos principales son el Magdalena y el Cauca.
1054:         La economÃa se basa en agricultura y ganaderÃa.
1055:         """
1056:
1057:         context = create_document_context(section='environment', chapter=4)
1058:
1059:         result = analyze_with_intelligence_layer(
1060:             text=irrelevant_doc,
1061:             signal_node=sample_question,
1062:             document_context=context
1063:         )
1064:
```

```
        print(f"\nâ\234\223 Irrelevant Content Filtering:")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Evidence extracted: {sum(len(v) for v in result['evidence'].values())}")

        # Should extract minimal or no evidence
        assert result['completeness'] < 0.3

    def test_04_document_with_conflicting_data(self, sample_question):
        """Test: Handle documents with conflicting information."""
        conflicting_doc = """
        LÃ-nea base segÃºn DANE: 8.5% de mujeres en cargos directivos.
        LÃ-nea base segÃºn SecretarÃ-a: 12.3% de mujeres en cargos directivos.

        Meta establecida: 15% para 2027.
        Otra meta: 20% para 2027.

        Presupuesto: COP 1,500 millones.
        Presupuesto revisado: COP 2,000 millones.
        """

        result = analyze_with_intelligence_layer(
            text=conflicting_doc,
            signal_node=sample_question,
            document_context={}
        )

        print(f"\nâ\234\223 Conflicting Data Handling:")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Evidence extracted: {sum(len(v) for v in result['evidence'].values())}")

        # Should extract evidence but may flag conflicts
        assert result['completeness'] > 0.0

    def test_05_document_with_special_characters(self, sample_question):
        """Test: Handle special characters and formatting."""
        special_chars_doc = """
        DIAGNÃ\223STICO: SituaciÃ³n de gÃ©nero en 2023

        LÃ-nea base (aÃ±o 2023): 8.5% â\206\222 Meta: 15% (Î\224 = +6.5%)
        Fuente: DANE Â© 2023

        Presupuesto: $1,500'000,000 COP
        Cobertura: 100% del territorio â\200¢ Todas las localidades

        Nota: Los datos incluyen informaciÃ³n de aÃ±os 2020â\200\2232023
        """

        result = analyze_with_intelligence_layer(
            text=special_chars_doc,
            signal_node=sample_question,
            document_context={}
        )

        print(f"\nâ\234\223 Special Characters Handling:")
        print(f"  Completeness: {result['completeness']:.2f}")

```

```
1121:         # Should handle special chars gracefully
1122:         assert result['completeness'] >= 0.0
1123:
1124:
1125: class TestContextAwareFiltering:
1126:     """Test context-aware pattern filtering effectiveness."""
1127:
1128:     def test_01_budget_context_filters_correctly(self, sample_question):
1129:         """Test: Budget context activates budget-specific patterns."""
1130:         patterns = sample_question.get('patterns', [])
1131:         base_pack = MockSignalPack(patterns, [sample_question])
1132:         enriched = create_enriched_signal_pack(base_pack, enable_semantic_expansion=False)
1133:
1134:         # Budget context
1135:         budget_ctx = create_document_context(section='budget', chapter=3)
1136:         budget_patterns = enriched.get_patterns_for_context(budget_ctx)
1137:
1138:         # Non-budget context
1139:         other_ctx = create_document_context(section='introduction', chapter=1)
1140:         other_patterns = enriched.get_patterns_for_context(other_ctx)
1141:
1142:         print(f"\nâ\234\223 Context-Aware Filtering:")
1143:         print(f"  Total patterns: {len(patterns)}")
1144:         print(f"  Budget context: {len(budget_patterns)} patterns")
1145:         print(f"  Other context: {len(other_patterns)} patterns")
1146:
1147:         # Both should have patterns (due to global patterns)
1148:         assert len(budget_patterns) > 0
1149:         assert len(other_patterns) > 0
1150:
1151:     def test_02_context_hierarchy_respected(self, sample_question):
1152:         """Test: Context scope hierarchy is respected."""
1153:         patterns = sample_question.get('patterns', [])
1154:
1155:         # Count by scope
1156:         scope_distribution = {}
1157:         for p in patterns:
1158:             scope = p.get('context_scope', 'UNKNOWN')
1159:             scope_distribution[scope] = scope_distribution.get(scope, 0) + 1
1160:
1161:         print(f"\nâ\234\223 Context Scope Distribution:")
1162:         for scope, count in sorted(scope_distribution.items()):
1163:             pct = (count / len(patterns) * 100) if patterns else 0
1164:             print(f"  {scope}: {count} patterns ({pct:.1f}%)")
1165:
1166:         # Verify scopes are defined
1167:         assert len(scope_distribution) > 0
1168:
1169:     def test_03_context_requirement_enforcement(self, sample_question):
1170:         """Test: Context requirements are enforced correctly."""
1171:         patterns = sample_question.get('patterns', [])
1172:
1173:         # Find patterns with requirements
1174:         patterns_with_req = [p for p in patterns if p.get('context_requirement')]
1175:
1176:         print(f"\nâ\234\223 Context Requirements:")
```

```
        print(f"  Patterns with requirements: {len(patterns_with_req)}/{len(patterns)}")

        if patterns_with_req:
            sample = patterns_with_req[0]
            print(f"  Example requirement: {sample.get('context_requirement')}")


class TestValidationContractScenarios:
    """Test validation contract scenarios."""

    def test_01_missing_required_elements_detected(self, sample_question):
        """Test: Missing required elements trigger validation failure."""
        expected = sample_question.get('expected_elements', [])
        required = [e for e in expected if isinstance(e, dict) and e.get('required')]

        if required:
            # Simulate incomplete extraction
            incomplete_result = {
                'completeness': 0.3,
                'missing_elements': [r.get('type', '') for r in required],
                'evidence': {}
            }

            from farfan_pipeline.core.orchestrator.signal_contract_validator import validate_with_contract

            validation = validate_with_contract(incomplete_result, sample_question)

            print(f"\nâ\234\223 Missing Required Elements:")
            print(f"  Required elements: {len(required)}")
            print(f"  Validation status: {validation.status}")
            print(f"  Passed: {validation.passed}")

    def test_02_validation_with_minimum_cardinality(self, sample_question):
        """Test: Minimum cardinality requirements validated."""
        expected = sample_question.get('expected_elements', [])
        with_minimum = [e for e in expected if isinstance(e, dict) and e.get('minimum')]

        if with_minimum:
            elem = with_minimum[0]
            elem_type = elem.get('type', '')
            minimum = elem.get('minimum', 0)

            print(f"\nâ\234\223 Minimum Cardinality Validation:")
            print(f"  Element: {elem_type}")
            print(f"  Minimum required: {minimum}")

            # Simulate under-minimum result
            under_result = extract_structured_evidence(
                text="Limited data.",
                signal_node=sample_question
            )

            print(f"  Under minimum: {under_result.under_minimum}")


class TestPipelinePerformance:
```

```
    """Test pipeline performance with realistic data volumes."""

    def test_01_large_document_processing(self, sample_question):
        """Test: Process large document efficiently."""
        # Create large document (simulated policy plan)
        large_doc_sections = [
            "CAPÃ\215TULO 1: DIAGNÃ\223STICO\n" + "AnÃ¡lisis detallado. " * 100,
            "CAPÃ\215TULO 2: OBJETIVOS\n" + "Objetivo estratÃ©gico. " * 100,
            "CAPÃ\215TULO 3: PRESUPUESTO\n" + "AsignaciÃ³n presupuestal. " * 100,
            "CAPÃ\215TULO 4: INDICADORES\n" + "Indicador de gestiÃ³n. " * 100,
        ]

        large_doc = "\n\n".join(large_doc_sections)

        print(f"\nâ\234\223 Large Document Processing:")
        print(f"  Document size: {len(large_doc)} characters")

        result = analyze_with_intelligence_layer(
            text=large_doc,
            signal_node=sample_question,
            document_context={}
        )

        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Evidence extracted: {sum(len(v) for v in result['evidence'].values())}")

        assert result is not None

    def test_02_multiple_pattern_matching(self, questionnaire):
        """Test: Efficiently match many patterns."""
        questions = questionnaire.get_micro_questions()

        # Get question with many patterns
        q_with_many_patterns = max(questions, key=lambda q: len(q.get('patterns', [])))
        pattern_count = len(q_with_many_patterns.get('patterns', []))

        print(f"\nâ\234\223 Multiple Pattern Matching:")
        print(f"  Pattern count: {pattern_count}")

        test_doc = "DANE reporta lÃ-nea base. Meta establecida. Presupuesto asignado."

        result = analyze_with_intelligence_layer(
            text=test_doc,
            signal_node=q_with_many_patterns,
            document_context={}
        )

        print(f"  Completeness: {result['completeness']:.2f}")


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


================================================================================
```

```
1289: FILE: tests/core/test_signal_registry_enriched.py
1290: ================================================================================
1291:
1292: """
1293: Tests for Enriched Signal Registry Factory (JOBFRONT 2)
1294:
1295: Verifies that create_enriched_signal_registry creates EnrichedSignalPacks
1296: without mutating the base QuestionnaireSignalRegistry.
1297: """
1298:
1299: import pytest
1300: from unittest.mock import Mock, patch, MagicMock
1301:
1302: from farfan_pipeline.core.orchestrator.factory import create_enriched_signal_registry
1303: from farfan_pipeline.core.orchestrator.signal_intelligence_layer import EnrichedSignalPack
1304:
1305:
1306: @pytest.fixture
1307: def mock_signal_registry():
1308:     """Mock QuestionnaireSignalRegistry for testing."""
1309:     mock_registry = Mock()
1310:
1311:     # Mock list_policy_areas to return known policy areas
1312:     mock_registry.list_policy_areas.return_value = ['PA01', 'PA02', 'PA03']
1313:
1314:     # Mock get() to return base signal packs
1315:     def mock_get(policy_area_id):
1316:         mock_pack = Mock()
1317:         mock_pack.patterns = [
1318:             {'id': f'{policy_area_id}_PAT001', 'pattern': 'test', 'confidence_weight': 0.8}
1319:         ]
1320:         mock_pack.micro_questions = []
1321:         return mock_pack
1322:
1323:     mock_registry.get = mock_get
1324:
1325:     return mock_registry
1326:
1327:
1328: @patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
1329: def test_create_enriched_signal_registry_returns_dict(mock_registry_class, mock_signal_registry):
1330:     """Test that create_enriched_signal_registry returns a dict."""
1331:     mock_registry_class.return_value = mock_signal_registry
1332:
1333:     registry = create_enriched_signal_registry(
1334:         monolith_path='/fake/path/questionnaire.json'
1335:     )
1336:
1337:     assert isinstance(registry, dict)
1338:     assert len(registry) > 0
1339:
1340:
1341: @patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
1342: def test_enriched_packs_are_created_for_each_policy_area(mock_registry_class, mock_signal_registry):
1343:     """Test that EnrichedSignalPack is created for each policy area."""
1344:     mock_registry_class.return_value = mock_signal_registry
```

```
1345:
1346:     registry = create_enriched_signal_registry(
1347:         monolith_path='/fake/path/questionnaire.json'
1348:     )
1349:
1350:     # Should have entries for PA01, PA02, PA03
1351:     assert 'PA01' in registry
1352:     assert 'PA02' in registry
1353:     assert 'PA03' in registry
1354:
1355:
1356: @patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
1357: def test_enriched_packs_are_enriched_signal_pack_instances(mock_registry_class, mock_signal_registry):
1358:     """Test that values in registry are EnrichedSignalPack instances."""
1359:     mock_registry_class.return_value = mock_signal_registry
1360:
1361:     registry = create_enriched_signal_registry(
1362:         monolith_path='/fake/path/questionnaire.json',
1363:         enable_semantic_expansion=False  # Disable to avoid expansion logic
1364:     )
1365:
1366:     for pa_id, pack in registry.items():
1367:         assert isinstance(pack, EnrichedSignalPack)
1368:
1369:
1370: @patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
1371: def test_semantic_expansion_flag_is_respected(mock_registry_class, mock_signal_registry):
1372:     """Test that enable_semantic_expansion flag is passed correctly."""
1373:     mock_registry_class.return_value = mock_signal_registry
1374:
1375:     # Test with expansion disabled
1376:     registry_no_expansion = create_enriched_signal_registry(
1377:         monolith_path='/fake/path/questionnaire.json',
1378:         enable_semantic_expansion=False
1379:     )
1380:
1381:     # Test with expansion enabled
1382:     registry_with_expansion = create_enriched_signal_registry(
1383:         monolith_path='/fake/path/questionnaire.json',
1384:         enable_semantic_expansion=True
1385:     )
1386:
1387:     # Both should succeed (actual expansion testing is in EnrichedSignalPack tests)
1388:     assert 'PA01' in registry_no_expansion
1389:     assert 'PA01' in registry_with_expansion
1390:
1391:
1392: @patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
1393: def test_handles_missing_signal_pack_gracefully(mock_registry_class):
1394:     """Test that None signal packs are skipped gracefully."""
1395:     mock_registry = Mock()
1396:     mock_registry.list_policy_areas.return_value = ['PA01', 'PA02', 'PA03']
1397:
1398:     # PA02 returns None
1399:     def mock_get(policy_area_id):
1400:         if policy_area_id == 'PA02':
```

```
            return None
        mock_pack = Mock(spec=['patterns', 'micro_questions'])
        mock_pack.patterns = [{'id': 'PAT001', 'pattern': 'test'}]
        mock_pack.micro_questions = []
        return mock_pack

    mock_registry.get = mock_get
    mock_registry_class.return_value = mock_registry

    registry = create_enriched_signal_registry(
        monolith_path='/fake/path/questionnaire.json',
        enable_semantic_expansion=False
    )

    # PA02 should be skipped
    assert 'PA01' in registry
    assert 'PA02' not in registry
    assert 'PA03' in registry


@patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
def test_uses_canonical_path_when_none(mock_registry_class, mock_signal_registry):
    """Test that None monolith_path uses canonical QUESTIONNAIRE_PATH."""
    mock_registry_class.return_value = mock_signal_registry

    registry = create_enriched_signal_registry(monolith_path=None)

    # Should have called QuestionnaireSignalRegistry with canonical path
    assert mock_registry_class.called
    args, kwargs = mock_registry_class.call_args
    assert len(args) > 0  # First arg should be path
    # The path should be a string (converted from QUESTIONNAIRE_PATH)
    assert isinstance(args[0], str)


@patch('farfan_pipeline.core.orchestrator.signal_registry.QuestionnaireSignalRegistry')
def test_base_registry_not_mutated(mock_registry_class, mock_signal_registry):
    """Test that base QuestionnaireSignalRegistry is not mutated."""
    mock_registry_class.return_value = mock_signal_registry

    # Create enriched registry
    enriched_registry = create_enriched_signal_registry(
        monolith_path='/fake/path/questionnaire.json',
        enable_semantic_expansion=False
    )

    # Verify base registry methods were called but not modified
    assert mock_signal_registry.list_policy_areas.called
    assert mock_signal_registry.get.called

    # Base registry should still work normally (not mutated)
    mock_signal_registry.list_policy_areas.reset_mock()
    policy_areas = mock_signal_registry.list_policy_areas()
    assert policy_areas == ['PA01', 'PA02', 'PA03']


```

```
1457: if __name__ == '__main__':
1458:     pytest.main([__file__, '-v'])
1459:
1460:
1461:
1462: ================================================================================
1463: FILE: tests/core/test_signal_resolution.py
1464: ================================================================================
1465:
1466: """Tests for Signal Resolution with Hard-Fail Semantics
1467:
1468: This module tests the signal resolution system including:
1469: - Resolution with all required signals available
1470: - Hard-fail behavior when signals are missing
1471: - Per-chunk caching in SignalRegistry
1472: """
1473:
1474: from unittest.mock import Mock
1475:
1476: import pytest
1477:
1478: from farfan_pipeline.core.orchestrator.signal_resolution import (
1479:     Chunk,
1480:     Question,
1481:     Signal,
1482:     _resolve_signals,
1483: )
1484: from farfan_pipeline.core.orchestrator.signals import SignalPack, SignalRegistry
1485:
1486: EXPECTED_TWO_SIGNALS = 2
1487:
1488:
1489: @pytest.fixture
1490: def mock_signal_registry():
1491:     """Create a mock signal registry for testing."""
1492:     registry = Mock()
1493:     registry._chunk_cache = {}
1494:     return registry
1495:
1496:
1497: @pytest.fixture
1498: def sample_chunk():
1499:     """Create a sample chunk for testing."""
1500:     return Chunk(chunk_id="chunk_001", text="Sample policy text")
1501:
1502:
1503: @pytest.fixture
1504: def sample_question_two_signals():
1505:     """Create a question requiring two signal types."""
1506:     return Question(question_id="Q001", signal_requirements={"budget", "actor"})
1507:
1508:
1509: @pytest.fixture
1510: def sample_question_one_signal():
1511:     """Create a question requiring one signal type."""
1512:     return Question(question_id="Q002", signal_requirements={"budget"})
```

```
1513:
1514:
1515: def test_resolve_signals_with_all_available(
1516:     sample_chunk, sample_question_two_signals, mock_signal_registry
1517: ):
1518:     """Test signal resolution succeeds when all required signals are available."""
1519:     budget_signal = Signal(signal_type="budget", content={"amount": 1000000})
1520:     actor_signal = Signal(signal_type="actor", content={"name": "Ministry of Finance"})
1521:
1522:     mock_signal_registry.get_signals_for_chunk.return_value = [
1523:         budget_signal,
1524:         actor_signal,
1525:     ]
1526:
1527:     result = _resolve_signals(
1528:         sample_chunk, sample_question_two_signals, mock_signal_registry
1529:     )
1530:
1531:     assert isinstance(result, tuple)
1532:     assert len(result) == EXPECTED_TWO_SIGNALS
1533:     assert budget_signal in result
1534:     assert actor_signal in result
1535:
1536:     mock_signal_registry.get_signals_for_chunk.assert_called_once_with(
1537:         sample_chunk, {"budget", "actor"}
1538:     )
1539:
1540:
1541: def test_resolve_signals_fails_on_missing(
1542:     sample_chunk, sample_question_two_signals, mock_signal_registry
1543: ):
1544:     """Test signal resolution fails with ValueError when signals are missing."""
1545:     budget_signal = Signal(signal_type="budget", content={"amount": 1000000})
1546:
1547:     mock_signal_registry.get_signals_for_chunk.return_value = [budget_signal]
1548:
1549:     with pytest.raises(ValueError) as exc_info:
1550:         _resolve_signals(
1551:             sample_chunk, sample_question_two_signals, mock_signal_registry
1552:         )
1553:
1554:     error_message = str(exc_info.value)
1555:     assert "Missing signals" in error_message
1556:     assert "actor" in error_message
1557:
1558:
1559: def test_resolve_signals_error_message_format(
1560:     sample_chunk, sample_question_two_signals, mock_signal_registry
1561: ):
1562:     """Test that error message correctly formats missing signal types."""
1563:     mock_signal_registry.get_signals_for_chunk.return_value = []
1564:
1565:     with pytest.raises(ValueError) as exc_info:
1566:         _resolve_signals(
1567:             sample_chunk, sample_question_two_signals, mock_signal_registry
1568:         )
```

```
1569:
1570:     error_message = str(exc_info.value)
1571:     assert "Missing signals" in error_message
1572:     assert "actor" in error_message or "budget" in error_message
1573:
1574:
1575: def test_resolve_signals_returns_immutable_tuple(
1576:     sample_chunk, sample_question_one_signal, mock_signal_registry
1577: ):
1578:     """Test that resolved signals are returned as an immutable tuple."""
1579:     budget_signal = Signal(signal_type="budget", content={"amount": 1000000})
1580:
1581:     mock_signal_registry.get_signals_for_chunk.return_value = [budget_signal]
1582:
1583:     result = _resolve_signals(
1584:         sample_chunk, sample_question_one_signal, mock_signal_registry
1585:     )
1586:
1587:     assert isinstance(result, tuple)
1588:     assert len(result) == 1
1589:     assert result[0] == budget_signal
1590:
1591:     with pytest.raises((TypeError, AttributeError)):
1592:         result[0] = Signal(signal_type="other", content={})
1593:
1594:
1595: def test_signal_registry_caching(mock_signal_registry, sample_chunk):
1596:     """Test that SignalRegistry caches signals per chunk."""
1597:     registry = SignalRegistry()
1598:
1599:     budget_pack = SignalPack(
1600:         version="1.0.0",
1601:         policy_area="PA01",
1602:         patterns=["budget", "allocation"],
1603:         source_fingerprint="abc123" * 8,
1604:     )
1605:     registry.put("budget", budget_pack)
1606:
1607:     chunk = Chunk(chunk_id="chunk_001", text="Sample text")
1608:     required_types = {"budget"}
1609:
1610:     signals_first = registry.get_signals_for_chunk(chunk, required_types)
1611:
1612:     assert len(signals_first) == 1
1613:     assert signals_first[0].signal_type == "budget"
1614:
1615:     signals_second = registry.get_signals_for_chunk(chunk, required_types)
1616:
1617:     assert len(signals_second) == 1
1618:     assert signals_second[0].signal_type == "budget"
1619:
1620:     assert chunk.chunk_id in registry._chunk_cache
1621:     assert signals_first == signals_second
1622:
1623:
1624: def test_signal_registry_cache_per_chunk(mock_signal_registry):
```

```
    """Test that cache is maintained separately for different chunks."""
    registry = SignalRegistry()

    budget_pack = SignalPack(
        version="1.0.0",
        policy_area="PA01",
        patterns=["budget"],
        source_fingerprint="abc123" * 8,
    )
    registry.put("budget", budget_pack)

    chunk1 = Chunk(chunk_id="chunk_001", text="Sample text 1")
    chunk2 = Chunk(chunk_id="chunk_002", text="Sample text 2")

    required_types = {"budget"}

    signals1 = registry.get_signals_for_chunk(chunk1, required_types)
    signals2 = registry.get_signals_for_chunk(chunk2, required_types)

    assert chunk1.chunk_id in registry._chunk_cache
    assert chunk2.chunk_id in registry._chunk_cache

    assert len(signals1) == 1
    assert len(signals2) == 1


def test_signal_registry_cache_cleared_on_clear():
    """Test that chunk cache is cleared when registry.clear() is called."""
    registry = SignalRegistry()

    budget_pack = SignalPack(
        version="1.0.0",
        policy_area="PA01",
        patterns=["budget"],
        source_fingerprint="abc123" * 8,
    )
    registry.put("budget", budget_pack)

    chunk = Chunk(chunk_id="chunk_001", text="Sample text")
    required_types = {"budget"}

    signals = registry.get_signals_for_chunk(chunk, required_types)
    assert len(signals) == 1
    assert chunk.chunk_id in registry._chunk_cache

    registry.clear()

    assert len(registry._chunk_cache) == 0


def test_resolve_signals_with_no_requirements(sample_chunk, mock_signal_registry):
    """Test signal resolution with empty requirements set."""
    question = Question(question_id="Q003", signal_requirements=set())

    mock_signal_registry.get_signals_for_chunk.return_value = []

```

```
    result = _resolve_signals(sample_chunk, question, mock_signal_registry)

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

    mock_signal_registry.get_signals_for_chunk.assert_called_once_with(
        sample_chunk, set()
    )


def test_resolve_signals_with_extra_signals_available(
    sample_chunk, sample_question_one_signal, mock_signal_registry
):
    """Test that extra signals beyond requirements don't cause issues."""
    budget_signal = Signal(signal_type="budget", content={"amount": 1000000})
    actor_signal = Signal(signal_type="actor", content={"name": "Ministry"})

    mock_signal_registry.get_signals_for_chunk.return_value = [
        budget_signal,
        actor_signal,
    ]

    result = _resolve_signals(
        sample_chunk, sample_question_one_signal, mock_signal_registry
    )

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


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


================================================================================
FILE: tests/core/test_spc_adapter.py
================================================================================

import pytest
from farfan_pipeline.utils.cpp_adapter import CPPAdapter, CPPAdapterError

class MockProvenance:
    def __init__(self, page_number=None, section_header=None):
        self.page_number = page_number
        self.section_header = section_header

class MockChunk:
    def __init__(self, id, text, text_span_start, policy_area_id=None, dimension_id=None):
        self.id = id
        self.text = text
        self.text_span = lambda: None
        self.text_span.start = text_span_start
        self.policy_area_id = policy_area_id
        self.dimension_id = dimension_id
        self.provenance = None
```

```
1737:
1738: class MockCanonPackage:
1739:     def __init__(self, chunks):
1740:         self.chunk_graph = lambda: None
1741:         self.chunk_graph.chunks = {chunk.id: chunk for chunk in chunks}
1742:
1743: def test_enforce_60_chunks():
1744:     adapter = CPPAdapter()
1745:
1746:     # Test with 59 chunks
1747:     chunks_59 = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10) for i in range(59)]
1748:     package_59 = MockCanonPackage(chunks_59)
1749:     with pytest.raises(CPPAdapterError, match="Cardinality mismatch: Expected 60 chunks"):
1750:         adapter.to_preprocessed_document(package_59, "doc1")
1751:
1752:     # Test with 61 chunks
1753:     chunks_61 = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10) for i in range(61)]
1754:     package_61 = MockCanonPackage(chunks_61)
1755:     with pytest.raises(CPPAdapterError, match="Cardinality mismatch: Expected 60 chunks"):
1756:         adapter.to_preprocessed_document(package_61, "doc1")
1757:
1758:     # Test with 60 chunks (should pass)
1759:     chunks_60 = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
1760:     for chunk in chunks_60:
1761:         chunk.chunk_type = "diagnostic"
1762:         chunk.provenance = MockProvenance(page_number=1, section_header="Section 1")
1763:     package_60 = MockCanonPackage(chunks_60)
1764:     adapter.to_preprocessed_document(package_60, "doc1")
1765:
1766: def test_enforce_metadata():
1767:     adapter = CPPAdapter()
1768:
1769:     # Test with missing policy_area_id
1770:     chunks_missing_pa = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, dimension_id="DIM01") for i in range(60)]
1771:     for chunk in chunks_missing_pa:
1772:         chunk.chunk_type = "diagnostic"
1773:         chunk.provenance = MockProvenance(page_number=1, section_header="Section 1")
1774:     package_missing_pa = MockCanonPackage(chunks_missing_pa)
1775:     with pytest.raises(CPPAdapterError, match="Missing policy_area_id"):
1776:         adapter.to_preprocessed_document(package_missing_pa, "doc1")
1777:
1778:     # Test with missing dimension_id
1779:     chunks_missing_dim = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01") for i in range(60)]
1780:     for chunk in chunks_missing_dim:
1781:         chunk.chunk_type = "diagnostic"
1782:         chunk.provenance = MockProvenance(page_number=1, section_header="Section 1")
1783:     package_missing_dim = MockCanonPackage(chunks_missing_dim)
1784:     with pytest.raises(CPPAdapterError, match="Missing dimension_id"):
1785:         adapter.to_preprocessed_document(package_missing_dim, "doc1")
1786:
1787:     # Test with missing chunk_type
1788:     chunks_missing_chunk_type = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(
60)]
1789:     package_missing_chunk_type = MockCanonPackage(chunks_missing_chunk_type)
1790:     with pytest.raises(CPPAdapterError, match="Missing chunk_type"):
1791:         adapter.to_preprocessed_document(package_missing_chunk_type, "doc1")
```

```
    # Test with invalid chunk_type
    chunks_invalid_chunk_type = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
    for chunk in chunks_invalid_chunk_type:
        chunk.chunk_type = "invalid"
    package_invalid_chunk_type = MockCanonPackage(chunks_invalid_chunk_type)
    with pytest.raises(CPPAdapterError, match="Invalid chunk_type"):
        adapter.to_preprocessed_document(package_invalid_chunk_type, "doc1")

def test_enforce_provenance():
    adapter = CPPAdapter()

    # Test with missing provenance
    chunks_missing_prov = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
    for chunk in chunks_missing_prov:
        chunk.chunk_type = "diagnostic"
    package_missing_prov = MockCanonPackage(chunks_missing_prov)
    with pytest.raises(CPPAdapterError, match="Missing provenance"):
        adapter.to_preprocessed_document(package_missing_prov, "doc1")

    # Test with missing page_number
    chunks_missing_page = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
    for chunk in chunks_missing_page:
        chunk.chunk_type = "diagnostic"
        chunk.provenance = MockProvenance(section_header="Section 1")
    package_missing_page = MockCanonPackage(chunks_missing_page)
    with pytest.raises(CPPAdapterError, match="Missing provenance.page_number"):
        adapter.to_preprocessed_document(package_missing_page, "doc1")

    # Test with missing section_header
    chunks_missing_section = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
    for chunk in chunks_missing_section:
        chunk.chunk_type = "diagnostic"
        chunk.provenance = MockProvenance(page_number=1)
    package_missing_section = MockCanonPackage(chunks_missing_section)
    with pytest.raises(CPPAdapterError, match="Missing provenance.section_header"):
        adapter.to_preprocessed_document(package_missing_section, "doc1")

def test_processing_mode():
    adapter = CPPAdapter()

    # Test with valid SPC
    chunks_valid = [MockChunk(id=f"c{i}", text=f"chunk {i}", text_span_start=i*10, policy_area_id="PA01", dimension_id="DIM01") for i in range(60)]
    for chunk in chunks_valid:
        chunk.chunk_type = "diagnostic"
        chunk.provenance = MockProvenance(page_number=1, section_header="Section 1")
    package_valid = MockCanonPackage(chunks_valid)
    doc = adapter.to_preprocessed_document(package_valid, "doc1")
    assert doc.processing_mode == "chunked"



================================================================================
FILE: tests/core/test_spc_adapter_integration.py
```

```
================================================================================

from farfan_pipeline.utils.cpp_adapter import CPPAdapter
from farfan_pipeline.core.phases.phase1_models import Chunk as CanonChunk
from farfan_pipeline.core.types import PreprocessedDocument


class MockProvenance:
    def __init__(self, page_number, section_header):
        self.page_number = page_number
        self.section_header = section_header


class MockCanonPolicyPackage:
    def __init__(self, chunks):
        self.chunk_graph = lambda: None
        self.chunk_graph.chunks = {chunk.id: chunk for chunk in chunks}
        self.schema_version = "SPC-2025.1"
        self.quality_metrics = None
        self.policy_manifest = None
        self.metadata = None


def create_mock_chunk(i):
    chunk = CanonChunk(
        id=f"c{i}",
        text=f"This is chunk {i}.",
        text_span_start=i * 20,
        policy_area_id=f"PA{i % 10 + 1:02d}",
        dimension_id=f"DIM{i % 6 + 1:02d}",
        chunk_type="diagnostic",
    )
    chunk.provenance = MockProvenance(
        page_number=i + 1, section_header=f"Section {i + 1}"
    )
    return chunk


def test_spc_adapter_integration():
    """
    Integration test for the SPC to PreprocessedDocument conversion.
    """
    # 1. Create a canonical test document with 60 chunks
    chunks = [create_mock_chunk(i) for i in range(60)]
    canon_package = MockCanonPolicyPackage(chunks)

    # 2. Run the adapter
    adapter = CPPAdapter()
    preprocessed_doc = adapter.to_preprocessed_document(canon_package, "test_doc")

    # 3. Verify the invariants
    assert isinstance(preprocessed_doc, PreprocessedDocument)
    assert preprocessed_doc.document_id == "test_doc"
    assert preprocessed_doc.processing_mode == "chunked"
    assert len(preprocessed_doc.chunks) == 60

```

```
    for i, chunk_data in enumerate(preprocessed_doc.chunks):
        assert chunk_data.policy_area_id is not None
        assert chunk_data.dimension_id is not None
        assert chunk_data.chunk_type is not None
        assert chunk_data.provenance is not None
        assert chunk_data.provenance.page_number is not None
        assert chunk_data.provenance.section_header is not None


================================================================================
FILE: tests/core/test_task_planner.py
================================================================================

import json
from pathlib import Path
from typing import Any

import pytest

from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
    ChunkRoutingResult,
)
from farfan_pipeline.core.orchestrator.task_planner import (
    EXPECTED_TASKS_PER_CHUNK,
    ExecutableTask,
    MicroQuestionContext,
    _construct_task,
    _construct_task_legacy,
    _validate_cross_task,
    _validate_schema,
    extract_expected_elements,
    extract_signal_requirements,
    sort_micro_question_contexts,
)
from farfan_pipeline.core.types import ChunkData


def create_test_chunk_routing_result(
    policy_area_id: str,
    chunk_id: str,
    dimension_id: str = "DIM01",
    text_content: str = "Test chunk content",
    expected_elements: list[dict[str, Any]] | None = None,
    document_position: tuple[int, int] | None = None,
) -> ChunkRoutingResult:
    """Helper function to create test ChunkRoutingResult with all required fields."""
    if expected_elements is None:
        expected_elements = []

    # Extract PA and DIM from the provided IDs for the chunk
    target_chunk = ChunkData(
        id=0,
        text=text_content,
        chunk_type="diagnostic",
        sentences=[],
```

```
1958:         tables=[],
1959:         start_pos=0,
1960:         end_pos=len(text_content),
1961:         confidence=0.95,
1962:         chunk_id=chunk_id,
1963:         policy_area_id=policy_area_id,
1964:         dimension_id=dimension_id,
1965:     )
1966:
1967:     return ChunkRoutingResult(
1968:         target_chunk=target_chunk,
1969:         chunk_id=chunk_id,
1970:         policy_area_id=policy_area_id,
1971:         dimension_id=dimension_id,
1972:         text_content=text_content,
1973:         expected_elements=expected_elements,
1974:         document_position=document_position,
1975:     )
1976:
1977:
1978: class TestValidateSchema:
1979:     def test_validate_schema_with_matching_elements(self):
1980:         question = {
1981:             "question_id": "Q001",
1982:             "expected_elements": [
1983:                 {"type": "fuentes_oficiales", "minimum": 2},
1984:                 {"type": "indicadores_cuantitativos", "minimum": 3},
1985:             ],
1986:         }
1987:         chunk = {
1988:             "id": "chunk_001",
1989:             "expected_elements": [
1990:                 {"type": "fuentes_oficiales", "minimum": 2},
1991:                 {"type": "indicadores_cuantitativos", "minimum": 3},
1992:             ],
1993:         }
1994:
1995:         _validate_schema(question, chunk)
1996:
1997:     def test_validate_schema_fails_on_mismatch(self):
1998:         question = {
1999:             "question_id": "Q001",
2000:             "expected_elements": [
2001:                 {"type": "fuentes_oficiales", "minimum": 2},
2002:                 {"type": "indicadores_cuantitativos", "minimum": 3},
2003:             ],
2004:         }
2005:         chunk = {
2006:             "id": "chunk_001",
2007:             "expected_elements": [
2008:                 {"type": "fuentes_oficiales", "minimum": 3},
2009:                 {"type": "series_temporales", "minimum": 2},
2010:             ],
2011:         }
2012:
2013:         with pytest.raises(ValueError) as exc_info:
```

```
            _validate_schema(question, chunk)

        assert "Schema mismatch" in str(exc_info.value)
        assert "fuentes_oficiales" in str(exc_info.value)
        assert "Question schema:" in str(exc_info.value)
        assert "Chunk schema:" in str(exc_info.value)

    def test_validate_schema_required_field_implication_passes_when_both_required(self):
        question = {
            "question_id": "Q001",
            "expected_elements": [
                {"type": "fuentes_oficiales", "required": True, "minimum": 2},
                {"type": "indicadores_cuantitativos", "required": True, "minimum": 3},
            ],
        }
        chunk = {
            "id": "chunk_001",
            "expected_elements": [
                {"type": "fuentes_oficiales", "required": True, "minimum": 2},
                {"type": "indicadores_cuantitativos", "required": True, "minimum": 3},
            ],
        }

        _validate_schema(question, chunk)

    def test_validate_schema_required_field_implication_passes_when_question_not_required(
        self,
    ):
        question = {
            "question_id": "Q001",
            "expected_elements": [
                {"type": "fuentes_oficiales", "required": False, "minimum": 2},
                {"type": "indicadores_cuantitativos", "minimum": 3},
            ],
        }
        chunk = {
            "id": "chunk_001",
            "expected_elements": [
                {"type": "fuentes_oficiales", "required": True, "minimum": 2},
                {"type": "indicadores_cuantitativos", "required": False, "minimum": 3},
            ],
        }

        _validate_schema(question, chunk)

    def test_validate_schema_required_field_implication_fails_when_question_required_but_chunk_not(
        self,
    ):
        question = {
            "question_id": "Q001",
            "expected_elements": [
                {"type": "fuentes_oficiales", "required": True, "minimum": 2},
                {"type": "indicadores_cuantitativos", "required": False, "minimum": 3},
            ],
        }
        chunk = {
```

```
2070:             "id": "chunk_001",
2071:             "expected_elements": [
2072:                 {"type": "fuentes_oficiales", "required": False, "minimum": 2},
2073:                 {"type": "indicadores_cuantitativos", "required": False, "minimum": 3},
2074:             ],
2075:         }
2076:
2077:         with pytest.raises(ValueError) as exc_info:
2078:             _validate_schema(question, chunk)
2079:
2080:         assert "Required-field implication violation" in str(exc_info.value)
2081:         assert "Q001" in str(exc_info.value)
2082:         assert "fuentes_oficiales" in str(exc_info.value)
2083:
2084:     def test_validate_schema_required_field_uses_false_as_default(self):
2085:         question = {
2086:             "question_id": "Q001",
2087:             "expected_elements": [
2088:                 {"type": "fuentes_oficiales", "minimum": 2},
2089:                 {"type": "indicadores_cuantitativos", "minimum": 3},
2090:             ],
2091:         }
2092:         chunk = {
2093:             "id": "chunk_001",
2094:             "expected_elements": [
2095:                 {"type": "fuentes_oficiales", "minimum": 2},
2096:                 {"type": "indicadores_cuantitativos", "minimum": 3},
2097:             ],
2098:         }
2099:
2100:         _validate_schema(question, chunk)
2101:
2102:
2103: class TestConstructTask:
2104:     def test_construct_task_generates_correct_id(self):
2105:         question = {
2106:             "question_id": "D1-Q1",
2107:             "question_global": 1,
2108:             "policy_area_id": "PA01",
2109:             "dimension_id": "DIM01",
2110:             "base_slot": "D1-Q1",
2111:             "cluster_id": "CL01",
2112:             "expected_elements": [{"type": "fuentes_oficiales", "minimum": 2}],
2113:             "signal_requirements": {"signal1": 0.3},
2114:         }
2115:         chunk = {"id": "chunk_001", "expected_elements": []}
2116:         patterns = [{"type": "pattern1"}]
2117:         signals = {"signal1": 0.5}
2118:         generated_task_ids: set[str] = set()
2119:
2120:         class MockRoutingResult:
2121:             policy_area_id = "PA01"
2122:
2123:         routing_result = MockRoutingResult()
2124:
2125:         task = _construct_task_legacy(
```

```
            question, chunk, patterns, signals, generated_task_ids, routing_result
        )

        assert task.task_id == "MQC-001_PA01"
        assert task.question_id == "D1-Q1"
        assert task.question_global == 1
        assert task.policy_area_id == "PA01"
        assert task.dimension_id == "DIM01"
        assert task.chunk_id == "chunk_001"
        assert task.patterns == patterns
        assert task.signals == signals
        assert task.expected_elements == question["expected_elements"]
        assert task.metadata["base_slot"] == "D1-Q1"
        assert task.metadata["cluster_id"] == "CL01"
        assert "MQC-001_PA01" in generated_task_ids

    def test_construct_task_rejects_duplicate_id(self):
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "policy_area_id": "PA01",
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001", "expected_elements": []}
        patterns = []
        signals = {}
        generated_task_ids = {"MQC-001_PA01"}

        class MockRoutingResult:
            policy_area_id = "PA01"

        routing_result = MockRoutingResult()

        with pytest.raises(ValueError) as exc_info:
            _construct_task_legacy(
                question, chunk, patterns, signals, generated_task_ids, routing_result
            )

        assert "Duplicate task_id detected: MQC-001_PA01 for question D1-Q1" in str(
            exc_info.value
        )

    def test_construct_task_timestamp_format(self):
        question = {
            "question_id": "D1-Q1",
            "question_global": 15,
            "policy_area_id": "PA05",
            "dimension_id": "DIM02",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_002", "expected_elements": []}
        generated_task_ids: set[str] = set()

        class MockRoutingResult:
            policy_area_id = "PA05"
```

```
2182:
2183:         routing_result = MockRoutingResult()
2184:
2185:         task = _construct_task_legacy(
2186:             question, chunk, [], {}, generated_task_ids, routing_result
2187:         )
2188:
2189:         assert "T" in task.creation_timestamp
2190:         assert task.creation_timestamp.endswith("Z") or "." in task.creation_timestamp
2191:
2192:
2193: class TestValidateCrossTask:
2194:     def test_cross_task_validation_with_canonical_questionnaire(self):
2195:         questionnaire_path = Path(
2196:             "system/config/questionnaire/questionnaire_monolith.json"
2197:         )
2198:
2199:         if not questionnaire_path.exists():
2200:             pytest.skip("Canonical questionnaire not found")
2201:
2202:         with open(questionnaire_path) as f:
2203:             data = json.load(f)
2204:
2205:         blocks = data.get("blocks", {})
2206:         micro_questions = blocks.get("micro_questions", [])
2207:
2208:         if not micro_questions:
2209:             pytest.skip("No micro_questions found in questionnaire")
2210:
2211:         plan: list[ExecutableTask] = []
2212:         generated_ids: set[str] = set()
2213:
2214:         chunk_map: dict[str, dict[str, Any]] = {}
2215:         for i in range(60):
2216:             chunk_id = f"chunk_{i:03d}"
2217:             chunk_map[chunk_id] = {"id": chunk_id, "expected_elements": []}
2218:
2219:         for idx, question in enumerate(micro_questions[:300]):
2220:             chunk_id = f"chunk_{(idx // 5):03d}"
2221:
2222:             task = ExecutableTask(
2223:                 task_id=f"MQC-{question.get('question_global', idx+1):03d}_{question.get('policy_area_id', 'PA01')}",
2224:                 question_id=question.get("question_id", f"Q{idx+1}"),
2225:                 question_global=question.get("question_global", idx + 1),
2226:                 policy_area_id=question.get("policy_area_id", "PA01"),
2227:                 dimension_id=question.get("dimension_id", "DIM01"),
2228:                 chunk_id=chunk_id,
2229:                 patterns=[],
2230:                 signals={},
2231:                 creation_timestamp="2024-01-01T00:00:00Z",
2232:                 expected_elements=[],
2233:                 metadata={},
2234:             )
2235:             plan.append(task)
2236:             generated_ids.add(task.task_id)
2237:
```

```
        _validate_cross_task(plan)

        chunk_usage: dict[str, int] = {}
        for task in plan:
            chunk_usage[task.chunk_id] = chunk_usage.get(task.chunk_id, 0) + 1

        for chunk_id in chunk_map:
            if chunk_id in chunk_usage:
                assert (
                    chunk_usage[chunk_id] == EXPECTED_TASKS_PER_CHUNK
                ), f"Chunk {chunk_id} should be used exactly {EXPECTED_TASKS_PER_CHUNK} times, got {chunk_usage[chunk_id]}"

    def test_cross_task_validation_warns_on_deviations(self, caplog):
        plan = [
            ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i}",
                question_global=i,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )
            for i in range(1, 4)
        ]

        with caplog.at_level("WARNING"):
            _validate_cross_task(plan)

        assert any(
            "Chunk usage deviation" in record.message for record in caplog.records
        )
        assert any(
            "chunk_001 used 3 times (expected 5)" in record.message
            for record in caplog.records
        )

    def test_cross_task_validation_policy_area_warning(self, caplog):
        plan = [
            ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i}",
                question_global=i,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id=f"chunk_{i:03d}",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )
```

```
            for i in range(1, 11)
        ]

        with caplog.at_level("WARNING"):
            _validate_cross_task(plan)

        assert any(
            "Policy area usage deviation" in record.message for record in caplog.records
        )
        assert any(
            "PA01 used 10 times (expected 30)" in record.message
            for record in caplog.records
        )


class TestExecutableTask:
    def test_executable_task_creation(self):
        signal_value = 0.8
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[{"type": "pattern1"}],
            signals={"signal1": signal_value},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[{"type": "test", "minimum": 1}],
            metadata={"key": "value"},
        )

        assert task.task_id == "MQC-001_PA01"
        assert task.question_global == 1
        assert task.policy_area_id == "PA01"
        assert len(task.patterns) == 1
        assert task.signals["signal1"] == signal_value
        assert len(task.expected_elements) == 1
        assert task.metadata["key"] == "value"


class TestMicroQuestionContext:
    def test_context_is_frozen(self):
        context = MicroQuestionContext(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            base_slot="D1-Q1",
            cluster_id="CL01",
            patterns=[{"type": "pattern1"}],
            signals={"signal1": 0.5},
            expected_elements=[{"type": "test", "minimum": 1}],
            signal_requirements={"signal1": 0.3},
```

```
2350:             creation_timestamp="2024-01-01T00:00:00Z",
2351:         )
2352:
2353:         assert context.task_id == "MQC-001_PA01"
2354:         assert context.question_global == 1
2355:         assert isinstance(context.patterns, tuple)
2356:         assert isinstance(context.expected_elements, tuple)
2357:
2358:         with pytest.raises(AttributeError):
2359:             context.task_id = "new_id"
2360:
2361:     def test_extract_expected_elements(self):
2362:         context = MicroQuestionContext(
2363:             task_id="MQC-001_PA01",
2364:             question_id="D1-Q1",
2365:             question_global=1,
2366:             policy_area_id="PA01",
2367:             dimension_id="DIM01",
2368:             chunk_id="chunk_001",
2369:             base_slot="D1-Q1",
2370:             cluster_id="CL01",
2371:             patterns=[],
2372:             signals={},
2373:             expected_elements=[{"type": "test", "minimum": 1}, {"type": "test2"}],
2374:             signal_requirements={},
2375:             creation_timestamp="2024-01-01T00:00:00Z",
2376:         )
2377:
2378:         elements = extract_expected_elements(context)
2379:         assert isinstance(elements, list)
2380:         assert len(elements) == 2
2381:         assert elements[0]["type"] == "test"
2382:
2383:     def test_extract_signal_requirements(self):
2384:         context = MicroQuestionContext(
2385:             task_id="MQC-001_PA01",
2386:             question_id="D1-Q1",
2387:             question_global=1,
2388:             policy_area_id="PA01",
2389:             dimension_id="DIM01",
2390:             chunk_id="chunk_001",
2391:             base_slot="D1-Q1",
2392:             cluster_id="CL01",
2393:             patterns=[],
2394:             signals={},
2395:             expected_elements=[],
2396:             signal_requirements={"signal1": 0.5, "signal2": 0.7},
2397:             creation_timestamp="2024-01-01T00:00:00Z",
2398:         )
2399:
2400:         requirements = extract_signal_requirements(context)
2401:         assert isinstance(requirements, dict)
2402:         assert requirements["signal1"] == 0.5
2403:         assert requirements["signal2"] == 0.7
2404:
2405:     def test_sort_micro_question_contexts(self):
```

```
2406:         contexts = [
2407:             MicroQuestionContext(
2408:                 task_id="MQC-002_PA02",
2409:                 question_id="D1-Q2",
2410:                 question_global=2,
2411:                 policy_area_id="PA02",
2412:                 dimension_id="DIM01",
2413:                 chunk_id="chunk_002",
2414:                 base_slot="D1-Q2",
2415:                 cluster_id="CL01",
2416:                 patterns=[],
2417:                 signals={},
2418:                 expected_elements=[],
2419:                 signal_requirements={},
2420:                 creation_timestamp="2024-01-01T00:00:00Z",
2421:             ),
2422:             MicroQuestionContext(
2423:                 task_id="MQC-051_PA01",
2424:                 question_id="D2-Q1",
2425:                 question_global=51,
2426:                 policy_area_id="PA01",
2427:                 dimension_id="DIM02",
2428:                 chunk_id="chunk_010",
2429:                 base_slot="D2-Q1",
2430:                 cluster_id="CL01",
2431:                 patterns=[],
2432:                 signals={},
2433:                 expected_elements=[],
2434:                 signal_requirements={},
2435:                 creation_timestamp="2024-01-01T00:00:00Z",
2436:             ),
2437:             MicroQuestionContext(
2438:                 task_id="MQC-001_PA01",
2439:                 question_id="D1-Q1",
2440:                 question_global=1,
2441:                 policy_area_id="PA01",
2442:                 dimension_id="DIM01",
2443:                 chunk_id="chunk_001",
2444:                 base_slot="D1-Q1",
2445:                 cluster_id="CL01",
2446:                 patterns=[],
2447:                 signals={},
2448:                 expected_elements=[],
2449:                 signal_requirements={},
2450:                 creation_timestamp="2024-01-01T00:00:00Z",
2451:             ),
2452:         ]
2453:
2454:         sorted_contexts = sort_micro_question_contexts(contexts)
2455:         assert sorted_contexts[0].question_global == 1
2456:         assert sorted_contexts[1].question_global == 2
2457:         assert sorted_contexts[2].question_global == 51
2458:         assert sorted_contexts[0].dimension_id == "DIM01"
2459:         assert sorted_contexts[2].dimension_id == "DIM02"
2460:
2461:
```

```
class TestConstructTaskNew:
    def test_construct_task_with_valid_inputs(self):
        question = {
            "question_id": "D1-Q1",
            "question_global": 42,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [{"type": "test", "minimum": 1}],
            "signal_requirements": {"signal1": 0.3},
        }
        routing_result = create_test_chunk_routing_result(
            policy_area_id="PA05", chunk_id="PA05-DIM01", dimension_id="DIM01"
        )
        applicable_patterns = ({"pattern": "p1"}, {"pattern": "p2"})
        resolved_signals = (0.8, 0.9, 0.7)
        generated_task_ids: set[str] = set()
        correlation_id = "corr-123-abc"

        task = _construct_task(
            question,
            routing_result,
            applicable_patterns,
            resolved_signals,
            generated_task_ids,
            correlation_id,
        )

        assert task.task_id == "MQC-042_PA05"
        assert task.question_id == "D1-Q1"
        assert task.question_global == 42
        assert task.policy_area_id == "PA05"
        assert task.dimension_id == "DIM01"
        assert task.chunk_id == "PA05-DIM01"
        assert len(task.patterns) == 2
        assert task.metadata["correlation_id"] == correlation_id
        assert task.metadata["base_slot"] == "D1-Q1"
        assert task.metadata["cluster_id"] == "CL01"
        assert "MQC-042_PA05" in generated_task_ids

    def test_construct_task_missing_question_global(self):
        question = {
            "question_id": "D1-Q1",
            "dimension_id": "DIM01",
        }
        routing_result = create_test_chunk_routing_result(
            policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
        )
        generated_task_ids: set[str] = set()

        with pytest.raises(ValueError) as exc_info:
            _construct_task(
                question, routing_result, (), (), generated_task_ids, "corr-123"
            )

        assert "Task construction failure for D1-Q1" in str(exc_info.value)
```

```
2518:         assert "question_global field missing or None" in str(exc_info.value)
2519:
2520:     def test_construct_task_missing_question_global_unknown_id(self):
2521:         question = {
2522:             "dimension_id": "DIM01",
2523:         }
2524:         routing_result = create_test_chunk_routing_result(
2525:             policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
2526:         )
2527:         generated_task_ids: set[str] = set()
2528:
2529:         with pytest.raises(ValueError) as exc_info:
2530:             _construct_task(
2531:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2532:             )
2533:
2534:         assert "Task construction failure for UNKNOWN" in str(exc_info.value)
2535:         assert "question_global field missing or None" in str(exc_info.value)
2536:
2537:     def test_construct_task_question_global_not_integer(self):
2538:         question = {
2539:             "question_id": "D1-Q1",
2540:             "question_global": "42",
2541:             "dimension_id": "DIM01",
2542:         }
2543:         routing_result = create_test_chunk_routing_result(
2544:             policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
2545:         )
2546:         generated_task_ids: set[str] = set()
2547:
2548:         with pytest.raises(ValueError) as exc_info:
2549:             _construct_task(
2550:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2551:             )
2552:
2553:         assert "Task construction failure for D1-Q1" in str(exc_info.value)
2554:         assert "question_global must be an integer" in str(exc_info.value)
2555:         assert "got str" in str(exc_info.value)
2556:
2557:     def test_construct_task_question_global_below_range(self):
2558:         question = {
2559:             "question_id": "D1-Q1",
2560:             "question_global": -1,
2561:             "dimension_id": "DIM01",
2562:         }
2563:         routing_result = create_test_chunk_routing_result(
2564:             policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
2565:         )
2566:         generated_task_ids: set[str] = set()
2567:
2568:         with pytest.raises(ValueError) as exc_info:
2569:             _construct_task(
2570:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2571:             )
2572:
2573:         assert "Task construction failure for D1-Q1" in str(exc_info.value)
```

```
2574:         assert "question_global must be in range 0-999" in str(exc_info.value)
2575:         assert "got -1" in str(exc_info.value)
2576:
2577:     def test_construct_task_question_global_above_range(self):
2578:         question = {
2579:             "question_id": "D1-Q1",
2580:             "question_global": 1000,
2581:             "dimension_id": "DIM01",
2582:         }
2583:         routing_result = create_test_chunk_routing_result(
2584:             policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
2585:         )
2586:         generated_task_ids: set[str] = set()
2587:
2588:         with pytest.raises(ValueError) as exc_info:
2589:             _construct_task(
2590:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2591:             )
2592:
2593:         assert "Task construction failure for D1-Q1" in str(exc_info.value)
2594:         assert "question_global must be in range 0-999" in str(exc_info.value)
2595:         assert "got 1000" in str(exc_info.value)
2596:
2597:     def test_construct_task_duplicate_task_id(self):
2598:         question = {
2599:             "question_id": "D1-Q1",
2600:             "question_global": 42,
2601:             "dimension_id": "DIM01",
2602:         }
2603:         routing_result = create_test_chunk_routing_result(
2604:             policy_area_id="PA05", chunk_id="PA05-DIM01", dimension_id="DIM01"
2605:         )
2606:         generated_task_ids = {"MQC-042_PA05"}
2607:
2608:         with pytest.raises(ValueError) as exc_info:
2609:             _construct_task(
2610:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2611:             )
2612:
2613:         assert "Duplicate task_id detected: MQC-042_PA05" in str(exc_info.value)
2614:
2615:     def test_construct_task_reserves_id_before_completion(self):
2616:         question = {
2617:             "question_id": "D1-Q1",
2618:             "question_global": 42,
2619:             "dimension_id": "DIM01",
2620:         }
2621:         routing_result = create_test_chunk_routing_result(
2622:             policy_area_id="PA05", chunk_id="PA05-DIM01", dimension_id="DIM01"
2623:         )
2624:         generated_task_ids: set[str] = set()
2625:
2626:         task = _construct_task(
2627:             question, routing_result, (), (), generated_task_ids, "corr-123"
2628:         )
2629:
```

```
        assert "MQC-042_PA05" in generated_task_ids
        assert task.task_id == "MQC-042_PA05"

    def test_construct_task_boundary_values(self):
        question_min = {
            "question_id": "D1-Q1",
            "question_global": 0,
            "dimension_id": "DIM01",
        }
        routing_result_min = create_test_chunk_routing_result(
            policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
        )
        generated_task_ids: set[str] = set()

        task_min = _construct_task(
            question_min, routing_result_min, (), (), generated_task_ids, "corr-123"
        )

        assert task_min.task_id == "MQC-000_PA01"
        assert task_min.question_global == 0

        question_max = {
            "question_id": "D6-Q50",
            "question_global": 999,
            "dimension_id": "DIM06",
        }
        routing_result_max = create_test_chunk_routing_result(
            policy_area_id="PA10", chunk_id="PA10-DIM06", dimension_id="DIM06"
        )

        task_max = _construct_task(
            question_max, routing_result_max, (), (), generated_task_ids, "corr-456"
        )

        assert task_max.task_id == "MQC-999_PA10"
        assert task_max.question_global == 999

    def test_construct_task_formats_task_id_with_leading_zeros(self):
        test_cases = [
            (1, "MQC-001_PA01"),
            (10, "MQC-010_PA01"),
            (100, "MQC-100_PA01"),
            (999, "MQC-999_PA01"),
        ]

        for question_global, expected_id in test_cases:
            question = {
                "question_id": f"Q{question_global}",
                "question_global": question_global,
                "dimension_id": "DIM01",
            }
            routing_result = create_test_chunk_routing_result(
                policy_area_id="PA01", chunk_id="PA01-DIM01", dimension_id="DIM01"
            )
            generated_task_ids: set[str] = set()

```

```
2686:             task = _construct_task(
2687:                 question, routing_result, (), (), generated_task_ids, "corr-123"
2688:             )
2689:
2690:             assert task.task_id == expected_id
2691:
2692:
2693:
2694: ================================================================================
2695: FILE: tests/flux/test_filter_patterns_comprehensive.py
2696: ================================================================================
2697:
2698: """
2699: Comprehensive Unit Tests for _filter_patterns()
2700: ================================================
2701:
2702: Test suite covering all aspects of pattern filtering including:
2703: - Exact policy_area_id match scenarios
2704: - Zero patterns after filtering (warning not error)
2705: - Pattern missing policy_area_id field (ValueError)
2706: - Pattern index construction and duplicate pattern_id handling
2707: - Immutability verification of returned tuple
2708: - Integration with validate_chunk_routing() and _construct_task()
2709: - Metadata tracking in task objects
2710:
2711: Test Standards:
2712: - pytest for test framework
2713: - hypothesis for property-based testing
2714: - Clear test names describing behavior
2715: - Comprehensive edge case coverage
2716:
2717: Coverage:
2718: - All primary branches of _filter_patterns()
2719: - All error paths and validation logic
2720: - Integration with task construction and routing
2721: - Edge cases and boundary conditions
2722: - Property-based guarantees
2723: """
2724:
2725: import logging
2726: from datetime import datetime, timezone
2727: from typing import Any
2728:
2729: import pytest
2730: from hypothesis import given
2731: from hypothesis import strategies as st
2732:
2733: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
2734:     IrrigationSynchronizer as CoreIrrigationSynchronizer,
2735: )
2736: from farfan_pipeline.core.orchestrator.task_planner import _construct_task
2737: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
2738: from farfan_pipeline.flux.irrigation_synchronizer import (
2739:     IrrigationSynchronizer,
2740:     Question,
2741: )
```

```
2742:
2743:
2744: @pytest.fixture
2745: def basic_question() -> Question:
2746:     """Create a basic question with valid patterns."""
2747:     return Question(
2748:         question_id="Q001",
2749:         policy_area_id="PA05",
2750:         dimension_id="D3",
2751:         patterns=[
2752:             {
2753:                 "pattern_id": "P001",
2754:                 "pattern": "test_pattern_1",
2755:                 "policy_area_id": "PA05",
2756:             },
2757:             {
2758:                 "pattern_id": "P002",
2759:                 "pattern": "test_pattern_2",
2760:                 "policy_area_id": "PA05",
2761:             },
2762:         ],
2763:     )
2764:
2765:
2766: @pytest.fixture
2767: def mixed_patterns_question() -> Question:
2768:     """Create question with patterns from multiple policy areas."""
2769:     return Question(
2770:         question_id="Q200",
2771:         policy_area_id="PA05",
2772:         dimension_id="D4",
2773:         patterns=[
2774:             {"pattern_id": "P1", "pattern": "pattern_pa01_1", "policy_area_id": "PA01"},
2775:             {"pattern_id": "P2", "pattern": "pattern_pa05_1", "policy_area_id": "PA05"},
2776:             {"pattern_id": "P3", "pattern": "pattern_pa02_1", "policy_area_id": "PA02"},
2777:             {"pattern_id": "P4", "pattern": "pattern_pa05_2", "policy_area_id": "PA05"},
2778:             {"pattern_id": "P5", "pattern": "pattern_pa01_2", "policy_area_id": "PA01"},
2779:             {"pattern_id": "P6", "pattern": "pattern_pa05_3", "policy_area_id": "PA05"},
2780:         ],
2781:     )
2782:
2783:
2784: @pytest.fixture
2785: def synchronizer() -> IrrigationSynchronizer:
2786:     """Create IrrigationSynchronizer instance."""
2787:     return IrrigationSynchronizer()
2788:
2789:
2790: class TestExactPolicyAreaMatch:
2791:     """Test exact policy_area_id matching scenarios."""
2792:
2793:     def test_exact_match_all_patterns_returned(
2794:         self, synchronizer: IrrigationSynchronizer, basic_question: Question
2795:     ) -> None:
2796:         """All patterns should be returned when they match target policy_area_id."""
2797:         filtered = synchronizer._filter_patterns(basic_question, "PA05")
```

```
2798:
2799:         assert isinstance(filtered, tuple), "Should return tuple"
2800:         assert len(filtered) == 2, "Should return all matching patterns"
2801:         assert all(p["policy_area_id"] == "PA05" for p in filtered)
2802:
2803:     def test_exact_match_preserves_pattern_data(
2804:         self, synchronizer: IrrigationSynchronizer, basic_question: Question
2805:     ) -> None:
2806:         """Pattern data should be preserved exactly during filtering."""
2807:         filtered = synchronizer._filter_patterns(basic_question, "PA05")
2808:
2809:         assert filtered[0]["pattern_id"] == "P001"
2810:         assert filtered[0]["pattern"] == "test_pattern_1"
2811:         assert filtered[1]["pattern_id"] == "P002"
2812:         assert filtered[1]["pattern"] == "test_pattern_2"
2813:
2814:     def test_exact_match_maintains_order(
2815:         self, synchronizer: IrrigationSynchronizer, mixed_patterns_question: Question
2816:     ) -> None:
2817:         """Filtered patterns should maintain original order."""
2818:         filtered = synchronizer._filter_patterns(mixed_patterns_question, "PA05")
2819:
2820:         assert len(filtered) == 3
2821:         assert filtered[0]["pattern_id"] == "P2"
2822:         assert filtered[1]["pattern_id"] == "P4"
2823:         assert filtered[2]["pattern_id"] == "P6"
2824:
2825:     def test_no_match_returns_empty_tuple(
2826:         self, synchronizer: IrrigationSynchronizer
2827:     ) -> None:
2828:         """Empty tuple should be returned when no patterns match."""
2829:         question = Question(
2830:             question_id="Q300",
2831:             policy_area_id="PA10",
2832:             dimension_id="D6",
2833:             patterns=[
2834:                 {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA01"},
2835:                 {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA02"},
2836:             ],
2837:         )
2838:
2839:         filtered = synchronizer._filter_patterns(question, "PA10")
2840:
2841:         assert isinstance(filtered, tuple)
2842:         assert len(filtered) == 0
2843:
2844:     def test_single_pattern_exact_match(
2845:         self, synchronizer: IrrigationSynchronizer
2846:     ) -> None:
2847:         """Single pattern matching should work correctly."""
2848:         question = Question(
2849:             question_id="Q400",
2850:             policy_area_id="PA03",
2851:             dimension_id="D2",
2852:             patterns=[
2853:                 {
```

```
                    "pattern_id": "P1",
                    "pattern": "solo_pattern",
                    "policy_area_id": "PA03",
                }
            ],
        )

        filtered = synchronizer._filter_patterns(question, "PA03")

        assert len(filtered) == 1
        assert filtered[0]["pattern_id"] == "P1"
        assert filtered[0]["policy_area_id"] == "PA03"


class TestZeroPatternsWarning:
    """Test zero patterns after filtering (warning not error)."""

    def test_zero_patterns_logs_warning(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Warning should be logged when zero patterns match."""
        question = Question(
            question_id="Q150",
            policy_area_id="PA07",
            dimension_id="D3",
            patterns=[
                {
                    "pattern_id": "P1",
                    "pattern": "pattern_pa01",
                    "policy_area_id": "PA01",
                },
                {
                    "pattern_id": "P2",
                    "pattern": "pattern_pa02",
                    "policy_area_id": "PA02",
                },
            ],
        )

        with caplog.at_level(logging.WARNING):
            result = synchronizer._filter_patterns(question, "PA07")

        assert len(result) == 0
        assert isinstance(result, tuple)
        assert any(
            "Zero patterns matched" in record.message for record in caplog.records
        )
        assert any("Q150" in record.message for record in caplog.records)
        assert any("PA07" in record.message for record in caplog.records)

    def test_zero_patterns_includes_context_in_warning(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Warning should include question_id, target PA, and pattern count."""
        question = Question(
            question_id="Q999",
```

```
            policy_area_id="PA08",
            dimension_id="D5",
            patterns=[
                {"pattern_id": "P1", "pattern": "p1", "policy_area_id": "PA01"},
                {"pattern_id": "P2", "pattern": "p2", "policy_area_id": "PA02"},
                {"pattern_id": "P3", "pattern": "p3", "policy_area_id": "PA03"},
            ],
        )

        with caplog.at_level(logging.WARNING):
            synchronizer._filter_patterns(question, "PA10")

        warning_msg = next(
            (r.message for r in caplog.records if r.levelname == "WARNING"), ""
        )
        assert "Q999" in warning_msg
        assert "PA10" in warning_msg
        assert "3" in warning_msg or "total patterns" in warning_msg.lower()

    def test_zero_patterns_does_not_raise_exception(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Zero patterns should not raise exception, only warn."""
        question = Question(
            question_id="Q500",
            policy_area_id="PA06",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA01"}
            ],
        )

        result = synchronizer._filter_patterns(question, "PA99")
        assert result == ()

    def test_empty_patterns_list_logs_warning(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Empty patterns list should also trigger warning."""
        question = Question(
            question_id="Q600",
            policy_area_id="PA01",
            dimension_id="D1",
            patterns=[],
        )

        with caplog.at_level(logging.WARNING):
            result = synchronizer._filter_patterns(question, "PA01")

        assert result == ()
        assert any(
            "Zero patterns matched" in record.message for record in caplog.records
        )


class TestMissingPolicyAreaIdField:
```

```
    """Test pattern missing policy_area_id field (ValueError)."""

    def test_missing_policy_area_id_raises_valueerror(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """ValueError should be raised when pattern lacks policy_area_id."""
        question = Question(
            question_id="Q123",
            policy_area_id="PA05",
            dimension_id="D3",
            patterns=[
                {"pattern_id": "P1", "pattern": "valid", "policy_area_id": "PA05"},
                {"pattern_id": "P2", "pattern": "invalid"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA05")

        error_msg = str(exc_info.value)
        assert "Q123" in error_msg
        assert "policy_area_id" in error_msg.lower()
        assert "index 1" in error_msg or "Pattern at index" in error_msg

    def test_missing_field_error_includes_question_id(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Error message should include question_id for debugging."""
        question = Question(
            question_id="Q789",
            policy_area_id="PA02",
            dimension_id="D4",
            patterns=[{"pattern_id": "P1", "pattern": "no_pa_field"}],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA02")

        assert "Q789" in str(exc_info.value)

    def test_missing_field_error_includes_pattern_index(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Error message should include pattern index."""
        question = Question(
            question_id="Q111",
            policy_area_id="PA03",
            dimension_id="D2",
            patterns=[
                {"pattern_id": "P1", "pattern": "valid1", "policy_area_id": "PA03"},
                {"pattern_id": "P2", "pattern": "valid2", "policy_area_id": "PA03"},
                {"pattern_id": "P3", "pattern": "invalid"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
```

```
            synchronizer._filter_patterns(question, "PA03")

        error_msg = str(exc_info.value)
        assert "index 2" in error_msg.lower() or "2" in error_msg

    def test_multiple_missing_fields_reports_first(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """When multiple patterns miss field, first occurrence should be reported."""
        question = Question(
            question_id="Q222",
            policy_area_id="PA04",
            dimension_id="D5",
            patterns=[
                {"pattern_id": "P1", "pattern": "valid", "policy_area_id": "PA04"},
                {"pattern_id": "P2", "pattern": "missing1"},
                {"pattern_id": "P3", "pattern": "missing2"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA04")

        error_msg = str(exc_info.value)
        assert "index 1" in error_msg.lower() or "1" in error_msg

    def test_validates_all_patterns_before_filtering(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Validation should happen before filtering logic."""
        question = Question(
            question_id="Q333",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "wrong_pa", "policy_area_id": "PA99"},
                {"pattern_id": "P2", "pattern": "missing"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA05")

        assert "missing" in str(exc_info.value).lower()


class TestPatternIndexConstruction:
    """Test pattern index construction and duplicate pattern_id handling."""

    def test_patterns_maintain_index_order(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Patterns should maintain their index order after filtering."""
        question = Question(
            question_id="Q700",
            policy_area_id="PA06",
```

```
3078:             dimension_id="D3",
3079:             patterns=[
3080:                 {"pattern_id": "P10", "pattern": "first", "policy_area_id": "PA06"},
3081:                 {"pattern_id": "P20", "pattern": "second", "policy_area_id": "PA06"},
3082:                 {"pattern_id": "P30", "pattern": "third", "policy_area_id": "PA06"},
3083:             ],
3084:         )
3085:
3086:         filtered = synchronizer._filter_patterns(question, "PA06")
3087:
3088:         assert filtered[0]["pattern_id"] == "P10"
3089:         assert filtered[1]["pattern_id"] == "P20"
3090:         assert filtered[2]["pattern_id"] == "P30"
3091:
3092:     def test_duplicate_pattern_ids_preserved(
3093:         self, synchronizer: IrrigationSynchronizer
3094:     ) -> None:
3095:         """Duplicate pattern_ids should be preserved (not deduplicated)."""
3096:         question = Question(
3097:             question_id="Q800",
3098:             policy_area_id="PA07",
3099:             dimension_id="D4",
3100:             patterns=[
3101:                 {"pattern_id": "P1", "pattern": "dup1", "policy_area_id": "PA07"},
3102:                 {"pattern_id": "P1", "pattern": "dup2", "policy_area_id": "PA07"},
3103:                 {"pattern_id": "P2", "pattern": "unique", "policy_area_id": "PA07"},
3104:             ],
3105:         )
3106:
3107:         filtered = synchronizer._filter_patterns(question, "PA07")
3108:
3109:         assert len(filtered) == 3
3110:         pattern_ids = [p["pattern_id"] for p in filtered]
3111:         assert pattern_ids.count("P1") == 2
3112:
3113:     def test_mixed_patterns_index_preserved(
3114:         self, synchronizer: IrrigationSynchronizer, mixed_patterns_question: Question
3115:     ) -> None:
3116:         """Index order should be preserved even with mixed policy areas."""
3117:         filtered = synchronizer._filter_patterns(mixed_patterns_question, "PA05")
3118:
3119:         expected_order = ["P2", "P4", "P6"]
3120:         actual_order = [p["pattern_id"] for p in filtered]
3121:         assert actual_order == expected_order
3122:
3123:     def test_patterns_without_pattern_id_field(
3124:         self, synchronizer: IrrigationSynchronizer
3125:     ) -> None:
3126:         """Patterns without pattern_id field should still be filterable."""
3127:         question = Question(
3128:             question_id="Q900",
3129:             policy_area_id="PA08",
3130:             dimension_id="D5",
3131:             patterns=[
3132:                 {"pattern": "no_id_1", "policy_area_id": "PA08"},
3133:                 {"pattern": "no_id_2", "policy_area_id": "PA08"},
```

```
            ],
        )

        filtered = synchronizer._filter_patterns(question, "PA08")

        assert len(filtered) == 2
        assert filtered[0]["pattern"] == "no_id_1"
        assert filtered[1]["pattern"] == "no_id_2"


class TestImmutabilityVerification:
    """Test immutability verification of returned tuple."""

    def test_returns_tuple_type(
        self, synchronizer: IrrigationSynchronizer, basic_question: Question
    ) -> None:
        """Return value must be tuple type."""
        result = synchronizer._filter_patterns(basic_question, "PA05")
        assert isinstance(result, tuple)
        assert type(result) is tuple

    def test_tuple_is_immutable(
        self, synchronizer: IrrigationSynchronizer, basic_question: Question
    ) -> None:
        """Returned tuple should not allow modification."""
        result = synchronizer._filter_patterns(basic_question, "PA05")

        with pytest.raises(AttributeError):
            result.append({"pattern_id": "P999", "policy_area_id": "PA05"})  # type: ignore

    def test_empty_result_is_tuple(self, synchronizer: IrrigationSynchronizer) -> None:
        """Empty result should also be tuple."""
        question = Question(
            question_id="Q1000",
            policy_area_id="PA09",
            dimension_id="D6",
            patterns=[
                {"pattern_id": "P1", "pattern": "other", "policy_area_id": "PA01"}
            ],
        )

        result = synchronizer._filter_patterns(question, "PA09")

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

    def test_multiple_calls_return_independent_tuples(
        self, synchronizer: IrrigationSynchronizer, basic_question: Question
    ) -> None:
        """Multiple calls should return independent tuple instances."""
        result1 = synchronizer._filter_patterns(basic_question, "PA05")
        result2 = synchronizer._filter_patterns(basic_question, "PA05")

        assert result1 is not result2
        assert result1 == result2

```

```
    def test_nested_dicts_not_protected(
        self, synchronizer: IrrigationSynchronizer, basic_question: Question
    ) -> None:
        """Note: tuple immutability doesn't prevent dict mutation (by design)."""
        result = synchronizer._filter_patterns(basic_question, "PA05")

        original_value = result[0]["pattern"]
        result[0]["pattern"] = "modified"

        assert result[0]["pattern"] == "modified"
        assert original_value == "test_pattern_1"


class TestIntegrationWithValidateChunkRouting:
    """Test integration with validate_chunk_routing()."""

    def create_complete_document(self) -> PreprocessedDocument:
        """Create valid document with all 60 chunks."""
        chunks = []
        chunk_id = 0
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(
                    ChunkData(
                        id=chunk_id,
                        text=f"Content for {pa_id} {dim_id}",
                        chunk_type="diagnostic",
                        sentences=[],
                        tables=[],
                        start_pos=0,
                        end_pos=20,
                        confidence=0.95,
                        policy_area_id=pa_id,
                        dimension_id=dim_id,
                    )
                )
                chunk_id += 1

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

    def test_validate_chunk_routing_with_filtered_patterns(self) -> None:
        """validate_chunk_routing should work with filtered patterns."""
        doc = self.create_complete_document()
        questionnaire = {"blocks": {}}
        core_sync = CoreIrrigationSynchronizer(
            questionnaire=questionnaire, preprocessed_document=doc
```

```
3246:         )
3247:
3248:         question = {
3249:             "question_id": "D1-Q01",
3250:             "policy_area_id": "PA05",
3251:             "dimension_id": "DIM03",
3252:             "patterns": [
3253:                 {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA05"},
3254:                 {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA05"},
3255:             ],
3256:         }
3257:
3258:         result = core_sync.validate_chunk_routing(question)
3259:
3260:         assert result.policy_area_id == "PA05"
3261:         assert result.dimension_id == "DIM03"
3262:         assert result.chunk_id == "PA05-DIM03"
3263:
3264:     def test_routing_result_contains_expected_fields(self) -> None:
3265:         """ChunkRoutingResult should contain all expected fields."""
3266:         doc = self.create_complete_document()
3267:         questionnaire = {"blocks": {}}
3268:         core_sync = CoreIrrigationSynchronizer(
3269:             questionnaire=questionnaire, preprocessed_document=doc
3270:         )
3271:
3272:         question = {
3273:             "question_id": "D2-Q05",
3274:             "policy_area_id": "PA07",
3275:             "dimension_id": "DIM04",
3276:             "patterns": [],
3277:             "expected_elements": [{"field": "value"}],
3278:         }
3279:
3280:         result = core_sync.validate_chunk_routing(question)
3281:
3282:         assert hasattr(result, "target_chunk")
3283:         assert hasattr(result, "chunk_id")
3284:         assert hasattr(result, "policy_area_id")
3285:         assert hasattr(result, "dimension_id")
3286:         assert hasattr(result, "text_content")
3287:         assert hasattr(result, "expected_elements")
3288:         assert result.expected_elements == [{"field": "value"}]
3289:
3290:
3291: class TestIntegrationWithConstructTask:
3292:     """Test integration with _construct_task()."""
3293:
3294:     def test_construct_task_with_filtered_patterns(self) -> None:
3295:         """_construct_task should accept filtered patterns."""
3296:         question = {
3297:             "question_id": "D1-Q01",
3298:             "question_global": 1,
3299:             "policy_area_id": "PA05",
3300:             "dimension_id": "DIM03",
3301:             "base_slot": "D1Q1",
```

```
            "cluster_id": "C1",
            "expected_elements": [],
            "signal_requirements": {},
        }

        chunk = {"id": "PA05-DIM03", "expected_elements": []}

        patterns = [
            {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA05"},
            {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA05"},
        ]

        signals = {}
        generated_ids: set[str] = set()

        task = _construct_task(question, chunk, patterns, signals, generated_ids)

        assert task.task_id == "MQC-001_PA05"
        assert task.patterns == patterns
        assert len(task.patterns) == 2

    def test_construct_task_with_empty_patterns(self) -> None:
        """_construct_task should handle empty patterns list."""
        question = {
            "question_id": "D2-Q10",
            "question_global": 60,
            "policy_area_id": "PA10",
            "dimension_id": "DIM06",
            "base_slot": "D2Q10",
            "cluster_id": "C10",
            "expected_elements": [],
            "signal_requirements": {},
        }

        chunk = {"id": "PA10-DIM06", "expected_elements": []}
        patterns: list[dict[str, Any]] = []
        signals = {}
        generated_ids: set[str] = set()

        task = _construct_task(question, chunk, patterns, signals, generated_ids)

        assert task.task_id == "MQC-060_PA10"
        assert task.patterns == []

    def test_construct_task_prevents_duplicate_task_ids(self) -> None:
        """_construct_task should raise ValueError on duplicate task_id."""
        question = {
            "question_id": "D1-Q01",
            "question_global": 1,
            "policy_area_id": "PA05",
            "dimension_id": "DIM03",
            "base_slot": "D1Q1",
            "cluster_id": "C1",
            "expected_elements": [],
            "signal_requirements": {},
        }
```

```
3358:
3359:         chunk = {"id": "PA05-DIM03", "expected_elements": []}
3360:         patterns: list[dict[str, Any]] = []
3361:         signals = {}
3362:         generated_ids = {"MQC-001_PA05"}
3363:
3364:         with pytest.raises(ValueError, match="Duplicate task_id detected"):
3365:             _construct_task(question, chunk, patterns, signals, generated_ids)
3366:
3367:
3368: class TestMetadataTracking:
3369:     """Test metadata tracking in task objects."""
3370:
3371:     def test_task_includes_pattern_metadata(self) -> None:
3372:         """Task should include pattern count in metadata or accessible via patterns."""
3373:         question = {
3374:             "question_id": "D3-Q15",
3375:             "question_global": 145,
3376:             "policy_area_id": "PA08",
3377:             "dimension_id": "DIM05",
3378:             "base_slot": "D3Q15",
3379:             "cluster_id": "C15",
3380:             "expected_elements": [],
3381:             "signal_requirements": {},
3382:         }
3383:
3384:         chunk = {"id": "PA08-DIM05", "expected_elements": []}
3385:
3386:         patterns = [
3387:             {"pattern_id": f"P{i}", "pattern": f"test{i}", "policy_area_id": "PA08"}
3388:             for i in range(5)
3389:         ]
3390:
3391:         signals = {}
3392:         generated_ids: set[str] = set()
3393:
3394:         task = _construct_task(question, chunk, patterns, signals, generated_ids)
3395:
3396:         assert len(task.patterns) == 5
3397:         assert task.metadata.get("base_slot") == "D3Q15"
3398:         assert task.metadata.get("cluster_id") == "C15"
3399:
3400:     def test_task_context_includes_immutable_patterns(self) -> None:
3401:         """Task context should include patterns as immutable tuple."""
3402:         question = {
3403:             "question_id": "D4-Q20",
3404:             "question_global": 200,
3405:             "policy_area_id": "PA03",
3406:             "dimension_id": "DIM02",
3407:             "base_slot": "D4Q20",
3408:             "cluster_id": "C20",
3409:             "expected_elements": [],
3410:             "signal_requirements": {},
3411:         }
3412:
3413:         chunk = {"id": "PA03-DIM02", "expected_elements": []}
```

```
3414:
3415:         patterns = [{"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA03"}]
3416:
3417:         signals = {}
3418:         generated_ids: set[str] = set()
3419:
3420:         task = _construct_task(question, chunk, patterns, signals, generated_ids)
3421:
3422:         assert task.context is not None
3423:         assert isinstance(task.context.patterns, tuple)
3424:         assert len(task.context.patterns) == 1
3425:
3426:     def test_task_creation_timestamp_recorded(self) -> None:
3427:         """Task should record creation timestamp."""
3428:         question = {
3429:             "question_id": "D5-Q25",
3430:             "question_global": 250,
3431:             "policy_area_id": "PA04",
3432:             "dimension_id": "DIM03",
3433:             "base_slot": "D5Q25",
3434:             "cluster_id": "C25",
3435:             "expected_elements": [],
3436:             "signal_requirements": {},
3437:         }
3438:
3439:         chunk = {"id": "PA04-DIM03", "expected_elements": []}
3440:         patterns: list[dict[str, Any]] = []
3441:         signals = {}
3442:         generated_ids: set[str] = set()
3443:
3444:         task = _construct_task(question, chunk, patterns, signals, generated_ids)
3445:
3446:         assert task.creation_timestamp is not None
3447:         assert isinstance(task.creation_timestamp, str)
3448:         assert "T" in task.creation_timestamp
3449:
3450:     def test_task_includes_expected_elements(self) -> None:
3451:         """Task should include expected_elements from question."""
3452:         question = {
3453:             "question_id": "D6-Q30",
3454:             "question_global": 300,
3455:             "policy_area_id": "PA09",
3456:             "dimension_id": "DIM06",
3457:             "base_slot": "D6Q30",
3458:             "cluster_id": "C30",
3459:             "expected_elements": [{"element_type": "indicator", "required": True}],
3460:             "signal_requirements": {},
3461:         }
3462:
3463:         chunk = {
3464:             "id": "PA09-DIM06",
3465:             "expected_elements": [{"element_type": "indicator", "required": True}],
3466:         }
3467:         patterns: list[dict[str, Any]] = []
3468:         signals = {}
3469:         generated_ids: set[str] = set()
```

```
3470:
3471:         task = _construct_task(question, chunk, patterns, signals, generated_ids)
3472:
3473:         assert len(task.expected_elements) == 1
3474:         assert task.expected_elements[0]["element_type"] == "indicator"
3475:
3476:
3477: class TestPropertyBasedFiltering:
3478:     """Property-based tests for pattern filtering guarantees."""
3479:
3480:     @given(
3481:         pa_count=st.integers(min_value=2, max_value=10),
3482:         patterns_per_pa=st.integers(min_value=1, max_value=20),
3483:         target_pa_idx=st.integers(min_value=0, max_value=9),
3484:     )
3485:     def test_no_cross_contamination(
3486:         self, pa_count: int, patterns_per_pa: int, target_pa_idx: int
3487:     ) -> None:
3488:         """No filtered pattern should contain different policy_area_id."""
3489:         if target_pa_idx >= pa_count:
3490:             target_pa_idx = target_pa_idx % pa_count
3491:
3492:         synchronizer = IrrigationSynchronizer()
3493:
3494:         all_patterns = []
3495:         policy_areas = [f"PA{i+1:02d}" for i in range(pa_count)]
3496:         target_pa = policy_areas[target_pa_idx]
3497:
3498:         for pa in policy_areas:
3499:             for i in range(patterns_per_pa):
3500:                 all_patterns.append(
3501:                     {
3502:                         "pattern_id": f"{pa}_P{i}",
3503:                         "pattern": f"pattern_{pa}_{i}",
3504:                         "policy_area_id": pa,
3505:                     }
3506:                 )
3507:
3508:         question = Question(
3509:             question_id="PROP_TEST",
3510:             policy_area_id=target_pa,
3511:             dimension_id="D1",
3512:             patterns=all_patterns,
3513:         )
3514:
3515:         filtered = synchronizer._filter_patterns(question, target_pa)
3516:
3517:         assert isinstance(filtered, tuple)
3518:         assert len(filtered) == patterns_per_pa
3519:
3520:         for pattern in filtered:
3521:             assert pattern["policy_area_id"] == target_pa
3522:
3523:     @given(
3524:         pattern_count=st.integers(min_value=0, max_value=100),
3525:     )
```

```
    def test_filtering_preserves_tuple_immutability(self, pattern_count: int) -> None:
        """Result should always be tuple regardless of pattern count."""
        synchronizer = IrrigationSynchronizer()

        patterns = [
            {
                "pattern_id": f"P{i}",
                "pattern": f"test_{i}",
                "policy_area_id": "PA05",
            }
            for i in range(pattern_count)
        ]

        question = Question(
            question_id="TUPLE_TEST",
            policy_area_id="PA05",
            dimension_id="D2",
            patterns=patterns,
        )

        result = synchronizer._filter_patterns(question, "PA05")

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

    @given(
        pattern_data=st.lists(
            st.tuples(
                st.text(min_size=1, max_size=10, alphabet="ABCDEFGHIJ"),
                st.text(min_size=1, max_size=20),
            ),
            min_size=1,
            max_size=50,
        )
    )
    def test_filtering_maintains_deterministic_order(
        self, pattern_data: list[tuple[str, str]]
    ) -> None:
        """Multiple calls should return same order."""
        synchronizer = IrrigationSynchronizer()

        patterns = [
            {
                "pattern_id": f"P{idx}",
                "pattern": text,
                "policy_area_id": pa_id,
            }
            for idx, (pa_id, text) in enumerate(pattern_data)
        ]

        question = Question(
            question_id="ORDER_TEST",
            policy_area_id="PA01",
            dimension_id="D3",
            patterns=patterns,
        )
```

```
3582:
3583:         result1 = synchronizer._filter_patterns(question, "A")
3584:         result2 = synchronizer._filter_patterns(question, "A")
3585:
3586:         assert result1 == result2
3587:
3588:
3589: class TestEdgeCasesAndBoundaryConditions:
3590:     """Test edge cases and boundary conditions."""
3591:
3592:     def test_unicode_in_pattern_fields(
3593:         self, synchronizer: IrrigationSynchronizer
3594:     ) -> None:
3595:         """Patterns with unicode characters should be handled correctly."""
3596:         question = Question(
3597:             question_id="Q_UNICODE",
3598:             policy_area_id="PA01",
3599:             dimension_id="D1",
3600:             patterns=[
3601:                 {"pattern_id": "P1", "pattern": "æµ\213è¯\225æ¨¡å¼\217", "policy_area_id": "PA01"},
3602:                 {"pattern_id": "P2", "pattern": "ã\203\221ã\202¿ã\203¼ã\203³", "policy_area_id": "PA01"},
3603:                 {"pattern_id": "P3", "pattern": "Ù\206Ù\205Ø•", "policy_area_id": "PA02"},
3604:             ],
3605:         )
3606:
3607:         filtered = synchronizer._filter_patterns(question, "PA01")
3608:
3609:         assert len(filtered) == 2
3610:         assert filtered[0]["pattern"] == "æµ\213è¯\225æ¨¡å¼\217"
3611:         assert filtered[1]["pattern"] == "ã\203\221ã\202¿ã\203¼ã\203³"
3612:
3613:     def test_special_characters_in_policy_area_id(
3614:         self, synchronizer: IrrigationSynchronizer
3615:     ) -> None:
3616:         """Policy area IDs with special format should work."""
3617:         question = Question(
3618:             question_id="Q_SPECIAL",
3619:             policy_area_id="PA01",
3620:             dimension_id="D1",
3621:             patterns=[
3622:                 {"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA01"},
3623:             ],
3624:         )
3625:
3626:         filtered = synchronizer._filter_patterns(question, "PA01")
3627:
3628:         assert len(filtered) == 1
3629:
3630:     def test_very_large_pattern_list(
3631:         self, synchronizer: IrrigationSynchronizer
3632:     ) -> None:
3633:         """Should handle very large pattern lists efficiently."""
3634:         patterns = [
3635:             {"pattern_id": f"P{i}", "pattern": f"test_{i}", "policy_area_id": "PA05"}
3636:             for i in range(1000)
3637:         ]
```

```
3638:
3639:         question = Question(
3640:             question_id="Q_LARGE",
3641:             policy_area_id="PA05",
3642:             dimension_id="D1",
3643:             patterns=patterns,
3644:         )
3645:
3646:         filtered = synchronizer._filter_patterns(question, "PA05")
3647:
3648:         assert len(filtered) == 1000
3649:         assert isinstance(filtered, tuple)
3650:
3651:     def test_pattern_with_none_values(
3652:         self, synchronizer: IrrigationSynchronizer
3653:     ) -> None:
3654:         """Patterns with None in non-policy_area_id fields should work."""
3655:         question = Question(
3656:             question_id="Q_NONE",
3657:             policy_area_id="PA03",
3658:             dimension_id="D2",
3659:             patterns=[
3660:                 {"pattern_id": None, "pattern": None, "policy_area_id": "PA03"},
3661:             ],
3662:         )
3663:
3664:         filtered = synchronizer._filter_patterns(question, "PA03")
3665:
3666:         assert len(filtered) == 1
3667:         assert filtered[0]["pattern_id"] is None
3668:
3669:     def test_pattern_with_nested_structures(
3670:         self, synchronizer: IrrigationSynchronizer
3671:     ) -> None:
3672:         """Patterns with nested dicts/lists should be preserved."""
3673:         question = Question(
3674:             question_id="Q_NESTED",
3675:             policy_area_id="PA04",
3676:             dimension_id="D3",
3677:             patterns=[
3678:                 {
3679:                     "pattern_id": "P1",
3680:                     "pattern": "test",
3681:                     "policy_area_id": "PA04",
3682:                     "metadata": {"key": "value", "nested": {"deep": "data"}},
3683:                     "tags": ["tag1", "tag2"],
3684:                 },
3685:             ],
3686:         )
3687:
3688:         filtered = synchronizer._filter_patterns(question, "PA04")
3689:
3690:         assert len(filtered) == 1
3691:         assert filtered[0]["metadata"]["nested"]["deep"] == "data"
3692:         assert filtered[0]["tags"] == ["tag1", "tag2"]
3693:
```

```
    def test_case_sensitive_policy_area_matching(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Policy area matching should be case-sensitive."""
        question = Question(
            question_id="Q_CASE",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "upper", "policy_area_id": "PA05"},
                {"pattern_id": "P2", "pattern": "lower", "policy_area_id": "pa05"},
            ],
        )

        filtered = synchronizer._filter_patterns(question, "PA05")

        assert len(filtered) == 1
        assert filtered[0]["pattern"] == "upper"

    def test_whitespace_in_policy_area_id(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Policy area IDs with whitespace should match exactly."""
        question = Question(
            question_id="Q_WHITESPACE",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "exact", "policy_area_id": "PA05"},
                {"pattern_id": "P2", "pattern": "spaces", "policy_area_id": " PA05 "},
            ],
        )

        filtered = synchronizer._filter_patterns(question, "PA05")

        assert len(filtered) == 1
        assert filtered[0]["pattern"] == "exact"

    def test_empty_string_policy_area_id(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Empty string policy area ID should be handled."""
        question = Question(
            question_id="Q_EMPTY",
            policy_area_id="",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "empty", "policy_area_id": ""},
                {"pattern_id": "P2", "pattern": "not_empty", "policy_area_id": "PA01"},
            ],
        )

        filtered = synchronizer._filter_patterns(question, "")

        assert len(filtered) == 1
        assert filtered[0]["pattern"] == "empty"
```

```
3750:
3751:     def test_numeric_policy_area_id(self, synchronizer: IrrigationSynchronizer) -> None:
3752:         """Numeric values for policy_area_id should be handled."""
3753:         question = Question(
3754:             question_id="Q_NUMERIC",
3755:             policy_area_id="PA01",
3756:             dimension_id="D1",
3757:             patterns=[
3758:                 {"pattern_id": "P1", "pattern": "string", "policy_area_id": "PA01"},
3759:                 {"pattern_id": "P2", "pattern": "number", "policy_area_id": 123},
3760:             ],
3761:         )
3762:
3763:         filtered = synchronizer._filter_patterns(question, "PA01")
3764:
3765:         assert len(filtered) == 1
3766:         assert filtered[0]["pattern"] == "string"
3767:
3768:
3769: class TestEndToEndPatternFilteringWorkflow:
3770:     """Test complete workflow from question to task with pattern filtering."""
3771:
3772:     def create_test_document(self) -> PreprocessedDocument:
3773:         """Create test document with all required chunks."""
3774:         chunks = []
3775:         chunk_id = 0
3776:         for pa_num in range(1, 11):
3777:             for dim_num in range(1, 7):
3778:                 pa_id = f"PA{pa_num:02d}"
3779:                 dim_id = f"DIM{dim_num:02d}"
3780:                 chunks.append(
3781:                     ChunkData(
3782:                         id=chunk_id,
3783:                         text=f"Test content for {pa_id} {dim_id}",
3784:                         chunk_type="diagnostic",
3785:                         sentences=[],
3786:                         tables=[],
3787:                         start_pos=0,
3788:                         end_pos=30,
3789:                         confidence=0.95,
3790:                         policy_area_id=pa_id,
3791:                         dimension_id=dim_id,
3792:                     )
3793:                 )
3794:                 chunk_id += 1
3795:
3796:         return PreprocessedDocument(
3797:             document_id="test-doc",
3798:             raw_text="test",
3799:             sentences=[],
3800:             tables=[],
3801:             metadata={},
3802:             chunks=chunks,
3803:             ingested_at=datetime.now(timezone.utc),
3804:         )
3805:
```

```
    def test_full_workflow_with_pattern_filtering(self) -> None:
        """Test complete workflow: question -> routing -> pattern filtering -> task."""
        doc = self.create_test_document()
        questionnaire = {"blocks": {}}
        core_sync = CoreIrrigationSynchronizer(
            questionnaire=questionnaire, preprocessed_document=doc
        )

        flux_sync = IrrigationSynchronizer()

        question_dict = {
            "question_id": "D1-Q05",
            "question_global": 5,
            "policy_area_id": "PA03",
            "dimension_id": "DIM02",
            "base_slot": "D1Q5",
            "cluster_id": "C5",
            "patterns": [
                {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA03"},
                {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA05"},
                {"pattern_id": "P3", "pattern": "test3", "policy_area_id": "PA03"},
            ],
            "expected_elements": [],
            "signal_requirements": {},
        }

        routing_result = core_sync.validate_chunk_routing(question_dict)

        flux_question = Question(
            question_id=question_dict["question_id"],
            policy_area_id=question_dict["policy_area_id"],
            dimension_id=question_dict["dimension_id"],
            patterns=question_dict["patterns"],
        )

        filtered_patterns = flux_sync._filter_patterns(flux_question, "PA03")

        assert len(filtered_patterns) == 2
        assert all(p["policy_area_id"] == "PA03" for p in filtered_patterns)

        chunk = {"id": routing_result.chunk_id, "expected_elements": []}
        patterns_list = list(filtered_patterns)
        generated_ids: set[str] = set()

        task = _construct_task(question_dict, chunk, patterns_list, {}, generated_ids)

        assert task.task_id == "MQC-005_PA03"
        assert len(task.patterns) == 2
        assert task.patterns[0]["pattern_id"] == "P1"
        assert task.patterns[1]["pattern_id"] == "P3"

    def test_pattern_filtering_across_multiple_questions(self) -> None:
        """Test pattern filtering consistency across multiple questions."""
        flux_sync = IrrigationSynchronizer()

        questions = [
```

```
            Question(
                question_id=f"Q{i}",
                policy_area_id=f"PA{(i % 10) + 1:02d}",
                dimension_id=f"D{(i % 6) + 1}",
                patterns=[
                    {
                        "pattern_id": f"P{j}",
                        "pattern": f"test_{j}",
                        "policy_area_id": f"PA{(j % 10) + 1:02d}",
                    }
                    for j in range(1, 11)
                ],
            )
            for i in range(30)
        ]

        for question in questions:
            filtered = flux_sync._filter_patterns(question, question.policy_area_id)

            assert all(p["policy_area_id"] == question.policy_area_id for p in filtered)
            assert len(filtered) > 0

    def test_filtered_patterns_used_in_task_context(self) -> None:
        """Verify filtered patterns are properly stored in task context."""
        question = {
            "question_id": "D3-Q10",
            "question_global": 140,
            "policy_area_id": "PA07",
            "dimension_id": "DIM04",
            "base_slot": "D3Q10",
            "cluster_id": "C10",
            "patterns": [
                {"pattern_id": "P1", "pattern": "keep1", "policy_area_id": "PA07"},
                {"pattern_id": "P2", "pattern": "drop", "policy_area_id": "PA01"},
                {"pattern_id": "P3", "pattern": "keep2", "policy_area_id": "PA07"},
            ],
            "expected_elements": [],
            "signal_requirements": {},
        }

        flux_sync = IrrigationSynchronizer()
        flux_question = Question(
            question_id=question["question_id"],
            policy_area_id=question["policy_area_id"],
            dimension_id=question["dimension_id"],
            patterns=question["patterns"],
        )

        filtered = flux_sync._filter_patterns(flux_question, "PA07")

        chunk = {"id": "PA07-DIM04", "expected_elements": []}
        patterns_list = list(filtered)
        generated_ids: set[str] = set()

        task = _construct_task(question, chunk, patterns_list, {}, generated_ids)

```

```
3918:         assert task.context is not None
3919:         assert len(task.context.patterns) == 2
3920:         assert task.context.patterns[0]["pattern_id"] == "P1"
3921:         assert task.context.patterns[1]["pattern_id"] == "P3"
3922:
3923:     def test_metadata_preserved_through_filtering(self) -> None:
3924:         """Verify all pattern metadata is preserved through filtering."""
3925:         question = {
3926:             "question_id": "D5-Q15",
3927:             "question_global": 245,
3928:             "policy_area_id": "PA09",
3929:             "dimension_id": "DIM05",
3930:             "base_slot": "D5Q15",
3931:             "cluster_id": "C15",
3932:             "patterns": [
3933:                 {
3934:                     "pattern_id": "P1",
3935:                     "pattern": "complex_pattern",
3936:                     "policy_area_id": "PA09",
3937:                     "confidence": 0.95,
3938:                     "source": "manual_annotation",
3939:                     "tags": ["important", "verified"],
3940:                     "metadata": {"created_by": "analyst_1", "version": 2},
3941:                 },
3942:             ],
3943:             "expected_elements": [],
3944:             "signal_requirements": {},
3945:         }
3946:
3947:         flux_sync = IrrigationSynchronizer()
3948:         flux_question = Question(
3949:             question_id=question["question_id"],
3950:             policy_area_id=question["policy_area_id"],
3951:             dimension_id=question["dimension_id"],
3952:             patterns=question["patterns"],
3953:         )
3954:
3955:         filtered = flux_sync._filter_patterns(flux_question, "PA09")
3956:
3957:         assert len(filtered) == 1
3958:         assert filtered[0]["confidence"] == 0.95
3959:         assert filtered[0]["source"] == "manual_annotation"
3960:         assert filtered[0]["tags"] == ["important", "verified"]
3961:         assert filtered[0]["metadata"]["created_by"] == "analyst_1"
3962:         assert filtered[0]["metadata"]["version"] == 2
3963:
3964:
3965: class TestLoggingBehavior:
3966:     """Test logging behavior of _filter_patterns()."""
3967:
3968:     def test_no_logging_on_successful_match(
3969:         self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
3970:     ) -> None:
3971:         """No warning should be logged when patterns match successfully."""
3972:         question = Question(
3973:             question_id="Q_SUCCESS",
```

```
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA05"}
            ],
        )

        with caplog.at_level(logging.WARNING):
            synchronizer._filter_patterns(question, "PA05")

        assert len(caplog.records) == 0

    def test_warning_logged_exactly_once(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Warning should be logged exactly once per zero-match call."""
        question = Question(
            question_id="Q_WARN",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA01"}
            ],
        )

        with caplog.at_level(logging.WARNING):
            synchronizer._filter_patterns(question, "PA05")
            synchronizer._filter_patterns(question, "PA05")

        warning_count = sum(
            1 for r in caplog.records if "Zero patterns matched" in r.message
        )
        assert warning_count == 2

    def test_warning_includes_pattern_count(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Warning should include total pattern count."""
        question = Question(
            question_id="Q_COUNT",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {
                    "pattern_id": f"P{i}",
                    "pattern": f"test_{i}",
                    "policy_area_id": "PA01",
                }
                for i in range(5)
            ],
        )

        with caplog.at_level(logging.WARNING):
            synchronizer._filter_patterns(question, "PA05")

        warning = next((r.message for r in caplog.records if "Zero" in r.message), "")
```

```
4030:         assert "5" in warning or "5 total patterns" in warning.lower()
4031:
4032:     def test_warning_includes_all_identifiers(
4033:         self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
4034:     ) -> None:
4035:         """Warning should include question_id, target PA, and question's PA."""
4036:         question = Question(
4037:             question_id="Q_IDENTIFIERS",
4038:             policy_area_id="PA08",
4039:             dimension_id="D3",
4040:             patterns=[
4041:                 {"pattern_id": "P1", "pattern": "test", "policy_area_id": "PA01"}
4042:             ],
4043:         )
4044:
4045:         with caplog.at_level(logging.WARNING):
4046:             synchronizer._filter_patterns(question, "PA10")
4047:
4048:         warning = next((r.message for r in caplog.records), "")
4049:         assert "Q_IDENTIFIERS" in warning
4050:         assert "PA10" in warning
4051:         assert "PA08" in warning
4052:
4053:
4054: class TestConcurrencyAndThreadSafety:
4055:     """Test thread-safety aspects of _filter_patterns()."""
4056:
4057:     def test_filter_patterns_is_stateless(
4058:         self, synchronizer: IrrigationSynchronizer
4059:     ) -> None:
4060:         """Multiple calls should not affect each other (stateless)."""
4061:         question1 = Question(
4062:             question_id="Q1",
4063:             policy_area_id="PA01",
4064:             dimension_id="D1",
4065:             patterns=[
4066:                 {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA01"}
4067:             ],
4068:         )
4069:
4070:         question2 = Question(
4071:             question_id="Q2",
4072:             policy_area_id="PA02",
4073:             dimension_id="D2",
4074:             patterns=[
4075:                 {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA02"}
4076:             ],
4077:         )
4078:
4079:         result1 = synchronizer._filter_patterns(question1, "PA01")
4080:         result2 = synchronizer._filter_patterns(question2, "PA02")
4081:         result1_again = synchronizer._filter_patterns(question1, "PA01")
4082:
4083:         assert result1 == result1_again
4084:         assert result1 != result2
4085:         assert result1[0]["pattern_id"] == "P1"
```

```
4086:         assert result2[0]["pattern_id"] == "P2"
4087:
4088:     def test_simultaneous_filtering_independence(
4089:         self, synchronizer: IrrigationSynchronizer
4090:     ) -> None:
4091:         """Results from one filtering operation shouldn't affect another."""
4092:         questions = [
4093:             Question(
4094:                 question_id=f"Q{i}",
4095:                 policy_area_id=f"PA{(i % 10) + 1:02d}",
4096:                 dimension_id=f"D{(i % 6) + 1}",
4097:                 patterns=[
4098:                     {
4099:                         "pattern_id": f"P{i}_{j}",
4100:                         "pattern": f"test_{i}_{j}",
4101:                         "policy_area_id": f"PA{(i % 10) + 1:02d}",
4102:                     }
4103:                     for j in range(3)
4104:                 ],
4105:             )
4106:             for i in range(10)
4107:         ]
4108:
4109:         results = [
4110:             synchronizer._filter_patterns(q, q.policy_area_id) for q in questions
4111:         ]
4112:
4113:         for i, result in enumerate(results):
4114:             assert len(result) == 3
4115:             assert all(p["pattern_id"].startswith(f"P{i}_") for p in result)
4116:
4117:
4118: class TestPerformanceCharacteristics:
4119:     """Test performance characteristics of _filter_patterns()."""
4120:
4121:     def test_linear_time_complexity(self, synchronizer: IrrigationSynchronizer) -> None:
4122:         """Filtering should have O(n) time complexity where n is pattern count."""
4123:         import time
4124:
4125:         sizes = [10, 100, 1000]
4126:         times = []
4127:
4128:         for size in sizes:
4129:             patterns = [
4130:                 {
4131:                     "pattern_id": f"P{i}",
4132:                     "pattern": f"test_{i}",
4133:                     "policy_area_id": "PA05",
4134:                 }
4135:                 for i in range(size)
4136:             ]
4137:
4138:             question = Question(
4139:                 question_id="Q_PERF",
4140:                 policy_area_id="PA05",
4141:                 dimension_id="D1",
```

```
                patterns=patterns,
            )

            start = time.perf_counter()
            for _ in range(100):
                synchronizer._filter_patterns(question, "PA05")
            elapsed = time.perf_counter() - start

            times.append(elapsed)

        assert times[0] > 0

    def test_filtering_does_not_modify_input(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Filtering should not modify input question or patterns."""
        original_patterns = [
            {"pattern_id": "P1", "pattern": "test1", "policy_area_id": "PA05"},
            {"pattern_id": "P2", "pattern": "test2", "policy_area_id": "PA01"},
        ]

        patterns_copy = [p.copy() for p in original_patterns]

        question = Question(
            question_id="Q_IMMUT",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=patterns_copy,
        )

        synchronizer._filter_patterns(question, "PA05")

        for i, pattern in enumerate(patterns_copy):
            assert pattern == original_patterns[i]


class TestRegressionTests:
    """Regression tests for previously found issues."""

    def test_empty_pattern_id_not_filtered_out(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Patterns with empty pattern_id should not be filtered out."""
        question = Question(
            question_id="Q_REGRESSION_1",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[{"pattern_id": "", "pattern": "test", "policy_area_id": "PA05"}],
        )

        filtered = synchronizer._filter_patterns(question, "PA05")

        assert len(filtered) == 1
        assert filtered[0]["pattern_id"] == ""

    def test_patterns_with_extra_fields_preserved(
```

```
4198:         self, synchronizer: IrrigationSynchronizer
4199:     ) -> None:
4200:         """Patterns with extra fields should have all fields preserved."""
4201:         question = Question(
4202:             question_id="Q_REGRESSION_2",
4203:             policy_area_id="PA05",
4204:             dimension_id="D1",
4205:             patterns=[
4206:                 {
4207:                     "pattern_id": "P1",
4208:                     "pattern": "test",
4209:                     "policy_area_id": "PA05",
4210:                     "extra_field_1": "value1",
4211:                     "extra_field_2": {"nested": "data"},
4212:                 }
4213:             ],
4214:         )
4215:
4216:         filtered = synchronizer._filter_patterns(question, "PA05")
4217:
4218:         assert filtered[0]["extra_field_1"] == "value1"
4219:         assert filtered[0]["extra_field_2"]["nested"] == "data"
4220:
4221:     def test_mixed_type_policy_area_ids_handled(
4222:         self, synchronizer: IrrigationSynchronizer
4223:     ) -> None:
4224:         """Mixed types in policy_area_id should be handled gracefully."""
4225:         question = Question(
4226:             question_id="Q_REGRESSION_3",
4227:             policy_area_id="PA05",
4228:             dimension_id="D1",
4229:             patterns=[
4230:                 {"pattern_id": "P1", "pattern": "str", "policy_area_id": "PA05"},
4231:                 {"pattern_id": "P2", "pattern": "int", "policy_area_id": 5},
4232:                 {"pattern_id": "P3", "pattern": "none", "policy_area_id": None},
4233:             ],
4234:         )
4235:
4236:         filtered = synchronizer._filter_patterns(question, "PA05")
4237:
4238:         assert len(filtered) == 1
4239:         assert filtered[0]["pattern"] == "str"
4240:
4241:
4242: class TestErrorHandlingAndValidation:
4243:     """Test comprehensive error handling and validation."""
4244:
4245:     def test_first_invalid_pattern_reported(
4246:         self, synchronizer: IrrigationSynchronizer
4247:     ) -> None:
4248:         """First invalid pattern should be reported when multiple are invalid."""
4249:         question = Question(
4250:             question_id="Q_FIRST_ERROR",
4251:             policy_area_id="PA05",
4252:             dimension_id="D1",
4253:             patterns=[
```

```
                {"pattern_id": "P1", "pattern": "valid", "policy_area_id": "PA05"},
                {"pattern_id": "P2", "pattern": "invalid_1"},
                {"pattern_id": "P3", "pattern": "invalid_2"},
                {"pattern_id": "P4", "pattern": "invalid_3"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA05")

        error = str(exc_info.value)
        assert "index 1" in error.lower()
        assert "P2" not in error or "invalid_1" not in error

    def test_validation_happens_before_filtering(
        self, synchronizer: IrrigationSynchronizer, caplog: pytest.LogCaptureFixture
    ) -> None:
        """Validation should happen before filtering (no warning logged)."""
        question = Question(
            question_id="Q_VALIDATE_FIRST",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "wrong_pa", "policy_area_id": "PA99"},
                {"pattern_id": "P2", "pattern": "no_pa"},
            ],
        )

        with caplog.at_level(logging.WARNING), pytest.raises(ValueError):
            synchronizer._filter_patterns(question, "PA05")

        assert len(caplog.records) == 0

    def test_all_patterns_validated_before_error(
        self, synchronizer: IrrigationSynchronizer
    ) -> None:
        """Validation should check in order and stop at first error."""
        question = Question(
            question_id="Q_VALIDATE_ORDER",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[
                {"pattern_id": "P1", "pattern": "valid1", "policy_area_id": "PA05"},
                {"pattern_id": "P2", "pattern": "valid2", "policy_area_id": "PA05"},
                {"pattern_id": "P3", "pattern": "valid3", "policy_area_id": "PA05"},
                {"pattern_id": "P4", "pattern": "invalid"},
                {"pattern_id": "P5", "pattern": "not_checked"},
            ],
        )

        with pytest.raises(ValueError) as exc_info:
            synchronizer._filter_patterns(question, "PA05")

        assert "index 3" in str(exc_info.value).lower()

    def test_error_message_format_consistency(
```

```
4310:         self, synchronizer: IrrigationSynchronizer
4311:     ) -> None:
4312:         """Error messages should have consistent format."""
4313:         questions_with_missing_field = [
4314:             Question(
4315:                 question_id=f"Q_{i}",
4316:                 policy_area_id=f"PA{(i % 10) + 1:02d}",
4317:                 dimension_id=f"D{(i % 6) + 1}",
4318:                 patterns=[{"pattern_id": "P1", "pattern": "test"}],
4319:             )
4320:             for i in range(5)
4321:         ]
4322:
4323:         for question in questions_with_missing_field:
4324:             with pytest.raises(ValueError) as exc_info:
4325:                 synchronizer._filter_patterns(question, question.policy_area_id)
4326:
4327:             error = str(exc_info.value)
4328:             assert "Pattern at index" in error
4329:             assert question.question_id in error
4330:             assert "policy_area_id" in error.lower()
4331:
4332:     def test_handles_get_method_gracefully(
4333:         self, synchronizer: IrrigationSynchronizer
4334:     ) -> None:
4335:         """Should use get() method which returns None for missing keys."""
4336:         question = Question(
4337:             question_id="Q_GET_METHOD",
4338:             policy_area_id="PA05",
4339:             dimension_id="D1",
4340:             patterns=[
4341:                 {"pattern_id": "P1", "pattern": "has_pa", "policy_area_id": "PA05"},
4342:             ],
4343:         )
4344:
4345:         filtered = synchronizer._filter_patterns(question, "PA05")
4346:         assert len(filtered) == 1
4347:
4348:
4349: class TestDocumentationAndTypeHints:
4350:     """Verify implementation matches documentation and type hints."""
4351:
4352:     def test_return_type_matches_signature(
4353:         self, synchronizer: IrrigationSynchronizer, basic_question: Question
4354:     ) -> None:
4355:         """Return type should match function signature (tuple)."""
4356:         result = synchronizer._filter_patterns(basic_question, "PA05")
4357:
4358:         assert isinstance(result, tuple)
4359:         assert all(isinstance(p, dict) for p in result)
4360:         assert all(isinstance(k, str) for p in result for k in p)
4361:
4362:     def test_raises_documented_exceptions(
4363:         self, synchronizer: IrrigationSynchronizer
4364:     ) -> None:
4365:         """Should raise ValueError as documented when validation fails."""
```

```
        question = Question(
            question_id="Q_DOC",
            policy_area_id="PA05",
            dimension_id="D1",
            patterns=[{"pattern_id": "P1", "pattern": "test"}],
        )

        with pytest.raises(ValueError):
            synchronizer._filter_patterns(question, "PA05")

    def test_function_has_correct_parameters(self) -> None:
        """Function should accept Question and str parameters."""
        import inspect

        sig = inspect.signature(IrrigationSynchronizer._filter_patterns)
        params = list(sig.parameters.keys())

        assert len(params) == 3
        assert params[0] == "self"
        assert params[1] == "question"
        assert params[2] == "target_pa_id"

    def test_immutable_return_documented_behavior(
        self, synchronizer: IrrigationSynchronizer, basic_question: Question
    ) -> None:
        """Immutable tuple return as documented in docstring."""
        result = synchronizer._filter_patterns(basic_question, "PA05")

        assert isinstance(result, tuple)

        try:
            result[0] = {"new": "pattern"}  # type: ignore
            raise AssertionError("Should not be able to modify tuple")
        except TypeError:
            pass


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

"""
Tests for IrrigationSynchronizer - Question-to-Chunk Matching
============================================================

Comprehensive test suite covering:
- O(1) chunk matching with complete and incomplete matrices
- Pattern filtering with validation and cross-contamination checks
- Property-based testing for pattern isolation guarantees

Test Standards:
- pytest for test framework
- hypothesis for property-based testing
- Clear test names describing behavior
"""
```

```
4422:
4423: import pytest
4424: from hypothesis import given, strategies as st
4425:
4426: from farfan_pipeline.flux.irrigation_synchronizer import (
4427:     ChunkMatrix,
4428:     Question,
4429:     IrrigationSynchronizer,
4430: )
4431:
4432:
4433: @pytest.fixture
4434: def complete_chunk_matrix() -> ChunkMatrix:
4435:     """Create a complete 10Ã\2276 matrix with all 60 chunks."""
4436:     chunks = {}
4437:     for pa_idx in range(1, 11):
4438:         pa_id = f"PA{pa_idx:02d}"
4439:         for dim_idx in range(1, 7):
4440:             dim_id = f"D{dim_idx}"
4441:             chunks[(pa_id, dim_id)] = {
4442:                 "chunk_id": f"{pa_id}_{dim_id}",
4443:                 "policy_area_id": pa_id,
4444:                 "dimension_id": dim_id,
4445:                 "text": f"Chunk for {pa_id} {dim_id}",
4446:             }
4447:     return ChunkMatrix(chunks=chunks)
4448:
4449:
4450: @pytest.fixture
4451: def incomplete_chunk_matrix() -> ChunkMatrix:
4452:     """Create matrix with 59 chunks (missing PA01_D1)."""
4453:     chunks = {}
4454:     for pa_idx in range(1, 11):
4455:         pa_id = f"PA{pa_idx:02d}"
4456:         for dim_idx in range(1, 7):
4457:             dim_id = f"D{dim_idx}"
4458:             if pa_id == "PA01" and dim_id == "D1":
4459:                 continue
4460:             chunks[(pa_id, dim_id)] = {
4461:                 "chunk_id": f"{pa_id}_{dim_id}",
4462:                 "policy_area_id": pa_id,
4463:                 "dimension_id": dim_id,
4464:                 "text": f"Chunk for {pa_id} {dim_id}",
4465:             }
4466:     return ChunkMatrix(chunks=chunks)
4467:
4468:
4469: @pytest.fixture
4470: def sample_questions() -> list[Question]:
4471:     """Create 300 questions (50 per dimension Ã\227 6 dimensions)."""
4472:     questions = []
4473:     q_idx = 1
4474:     for dim_idx in range(1, 7):
4475:         dim_id = f"D{dim_idx}"
4476:         for pa_idx in range(1, 11):
4477:             pa_id = f"PA{pa_idx:02d}"
```

```
            for q_in_pa in range(5):
                questions.append(
                    Question(
                        question_id=f"Q{q_idx:03d}",
                        policy_area_id=pa_id,
                        dimension_id=dim_id,
                        patterns=[
                            {
                                "pattern": f"pattern_{q_idx}",
                                "policy_area_id": pa_id,
                            }
                        ],
                    )
                )
                q_idx += 1
    return questions


def test_match_chunk_with_complete_matrix(
    complete_chunk_matrix: ChunkMatrix,
    sample_questions: list[Question]
) -> None:
    """Test _match_chunk succeeds with complete 60-chunk matrix and 300 questions."""
    synchronizer = IrrigationSynchronizer()

    assert len(sample_questions) == 300, "Should have 300 questions"
    assert len(complete_chunk_matrix.chunks) == 60, "Should have 60 chunks"

    for question in sample_questions:
        chunk = synchronizer._match_chunk(question, complete_chunk_matrix)

        assert chunk is not None
        assert chunk["policy_area_id"] == question.policy_area_id
        assert chunk["dimension_id"] == question.dimension_id
        assert chunk["chunk_id"] == f"{question.policy_area_id}_{question.dimension_id}"


def test_match_chunk_fails_on_missing_chunk(
    incomplete_chunk_matrix: ChunkMatrix
) -> None:
    """Test _match_chunk raises ValueError with descriptive message for missing chunk."""
    synchronizer = IrrigationSynchronizer()

    question = Question(
        question_id="Q001",
        policy_area_id="PA01",
        dimension_id="D1",
        patterns=[],
    )

    with pytest.raises(ValueError) as exc_info:
        synchronizer._match_chunk(question, incomplete_chunk_matrix)

    error_msg = str(exc_info.value)
    assert "Q001" in error_msg, "Error should include question_id"
    assert "PA01" in error_msg, "Error should include policy_area_id"
```

```
4534:     assert "D1" in error_msg, "Error should include dimension_id"
4535:     assert "No chunk found" in error_msg or "Failed to match" in error_msg
4536:
4537:
4538: def test_filter_patterns_enforces_policy_area_id_field() -> None:
4539:     """Test _filter_patterns validates all patterns have 'policy_area_id' field."""
4540:     synchronizer = IrrigationSynchronizer()
4541:
4542:     question_with_invalid_pattern = Question(
4543:         question_id="Q123",
4544:         policy_area_id="PA05",
4545:         dimension_id="D3",
4546:         patterns=[
4547:             {"pattern": "valid_pattern", "policy_area_id": "PA05"},
4548:             {"pattern": "invalid_pattern"},
4549:         ],
4550:     )
4551:
4552:     with pytest.raises(ValueError) as exc_info:
4553:         synchronizer._filter_patterns(question_with_invalid_pattern, "PA05")
4554:
4555:     error_msg = str(exc_info.value)
4556:     assert "Q123" in error_msg, "Error should include question_id"
4557:     assert "policy_area_id" in error_msg.lower(), "Error should mention missing field"
4558:     assert "index 1" in error_msg or "Pattern at index" in error_msg
4559:
4560:
4561: def test_filter_patterns_returns_only_matching_pa() -> None:
4562:     """Test _filter_patterns with mixed PA01/PA02/PA05 patterns filtering to PA05."""
4563:     synchronizer = IrrigationSynchronizer()
4564:
4565:     question = Question(
4566:         question_id="Q200",
4567:         policy_area_id="PA05",
4568:         dimension_id="D4",
4569:         patterns=[
4570:             {"pattern": "pattern_pa01_1", "policy_area_id": "PA01"},
4571:             {"pattern": "pattern_pa05_1", "policy_area_id": "PA05"},
4572:             {"pattern": "pattern_pa02_1", "policy_area_id": "PA02"},
4573:             {"pattern": "pattern_pa05_2", "policy_area_id": "PA05"},
4574:             {"pattern": "pattern_pa01_2", "policy_area_id": "PA01"},
4575:             {"pattern": "pattern_pa05_3", "policy_area_id": "PA05"},
4576:         ],
4577:     )
4578:
4579:     filtered = synchronizer._filter_patterns(question, "PA05")
4580:
4581:     assert isinstance(filtered, tuple), "Should return tuple for immutability"
4582:     assert len(filtered) == 3, "Should have exactly 3 PA05 patterns"
4583:
4584:     for pattern in filtered:
4585:         assert pattern["policy_area_id"] == "PA05", "All patterns should be PA05"
4586:
4587:     expected_patterns = {"pattern_pa05_1", "pattern_pa05_2", "pattern_pa05_3"}
4588:     actual_patterns = {p["pattern"] for p in filtered}
4589:     assert actual_patterns == expected_patterns, "Should have correct patterns"
```

```
4590:
4591:
4592: def test_filter_patterns_returns_immutable_tuple() -> None:
4593:     """Test that _filter_patterns returns tuple type for immutability."""
4594:     synchronizer = IrrigationSynchronizer()
4595:
4596:     question = Question(
4597:         question_id="Q300",
4598:         policy_area_id="PA10",
4599:         dimension_id="D6",
4600:         patterns=[
4601:             {"pattern": "test_pattern", "policy_area_id": "PA10"},
4602:         ],
4603:     )
4604:
4605:     result = synchronizer._filter_patterns(question, "PA10")
4606:
4607:     assert isinstance(result, tuple), "Return type must be tuple"
4608:
4609:     with pytest.raises(AttributeError):
4610:         result.append({"pattern": "new_pattern", "policy_area_id": "PA10"})  # type: ignore
4611:
4612:
4613: def test_filter_patterns_logs_warning_on_zero_matches(caplog: pytest.LogCaptureFixture) -> None:
4614:     """Test _filter_patterns logs warning when zero patterns match but does not fail."""
4615:     synchronizer = IrrigationSynchronizer()
4616:
4617:     question = Question(
4618:         question_id="Q150",
4619:         policy_area_id="PA07",
4620:         dimension_id="D3",
4621:         patterns=[
4622:             {"pattern": "pattern_pa01", "policy_area_id": "PA01"},
4623:             {"pattern": "pattern_pa02", "policy_area_id": "PA02"},
4624:         ],
4625:     )
4626:
4627:     with caplog.at_level("WARNING"):
4628:         result = synchronizer._filter_patterns(question, "PA07")
4629:
4630:     assert len(result) == 0, "Should return empty tuple"
4631:     assert isinstance(result, tuple), "Should still return tuple"
4632:
4633:     assert any("Zero patterns matched" in record.message for record in caplog.records)
4634:     assert any("Q150" in record.message for record in caplog.records)
4635:     assert any("PA07" in record.message for record in caplog.records)
4636:
4637:
4638: @given(
4639:     pa_count=st.integers(min_value=2, max_value=10),
4640:     patterns_per_pa=st.integers(min_value=1, max_value=20),
4641:     target_pa_idx=st.integers(min_value=0, max_value=9),
4642: )
4643: def test_filter_patterns_no_cross_contamination(
4644:     pa_count: int,
4645:     patterns_per_pa: int,
```

```
    target_pa_idx: int,
) -> None:
    """Property test verifying no filtered pattern contains different policy_area_id."""
    if target_pa_idx >= pa_count:
        target_pa_idx = target_pa_idx % pa_count

    synchronizer = IrrigationSynchronizer()

    all_patterns = []
    policy_areas = [f"PA{i+1:02d}" for i in range(pa_count)]
    target_pa = policy_areas[target_pa_idx]

    for pa in policy_areas:
        for i in range(patterns_per_pa):
            all_patterns.append({
                "pattern": f"pattern_{pa}_{i}",
                "policy_area_id": pa,
            })

    question = Question(
        question_id="PROP_TEST",
        policy_area_id=target_pa,
        dimension_id="D1",
        patterns=all_patterns,
    )

    filtered = synchronizer._filter_patterns(question, target_pa)

    assert isinstance(filtered, tuple), "Must return tuple"
    assert len(filtered) == patterns_per_pa, f"Should have {patterns_per_pa} patterns"

    for pattern in filtered:
        assert pattern["policy_area_id"] == target_pa, (
            f"Cross-contamination detected: pattern has policy_area_id="
            f"'{pattern['policy_area_id']}' but expected '{target_pa}'"
        )

    non_target_patterns = [p for p in all_patterns if p["policy_area_id"] != target_pa]
    filtered_set = {p["pattern"] for p in filtered}
    for non_target in non_target_patterns:
        assert non_target["pattern"] not in filtered_set, (
            f"Non-target pattern '{non_target['pattern']}' leaked into filtered results"
        )


def test_chunk_matrix_get_chunk_success() -> None:
    """Test ChunkMatrix.get_chunk succeeds for existing chunk."""
    chunks = {
        ("PA01", "D1"): {"chunk_id": "PA01_D1"},
        ("PA02", "D3"): {"chunk_id": "PA02_D3"},
    }
    matrix = ChunkMatrix(chunks=chunks)

    chunk = matrix.get_chunk("PA01", "D1")
    assert chunk["chunk_id"] == "PA01_D1"

```

```
    chunk = matrix.get_chunk("PA02", "D3")
    assert chunk["chunk_id"] == "PA02_D3"


def test_chunk_matrix_get_chunk_raises_on_missing() -> None:
    """Test ChunkMatrix.get_chunk raises ValueError for missing chunk."""
    chunks = {
        ("PA01", "D1"): {"chunk_id": "PA01_D1"},
    }
    matrix = ChunkMatrix(chunks=chunks)

    with pytest.raises(ValueError) as exc_info:
        matrix.get_chunk("PA99", "D9")

    error_msg = str(exc_info.value)
    assert "PA99" in error_msg
    assert "D9" in error_msg
    assert "No chunk found" in error_msg


def test_complete_workflow_300_questions_60_chunks(
    complete_chunk_matrix: ChunkMatrix,
    sample_questions: list[Question]
) -> None:
    """Integration test: complete workflow with 300 questions and 60 chunks."""
    synchronizer = IrrigationSynchronizer()

    results = []
    for question in sample_questions:
        chunk = synchronizer._match_chunk(question, complete_chunk_matrix)

        filtered_patterns = synchronizer._filter_patterns(
            question,
            question.policy_area_id
        )

        results.append({
            "question_id": question.question_id,
            "chunk_id": chunk["chunk_id"],
            "pattern_count": len(filtered_patterns),
        })

    assert len(results) == 300, "Should process all 300 questions"

    unique_chunks = {r["chunk_id"] for r in results}
    assert len(unique_chunks) == 60, "Should access all 60 unique chunks"

    for result in results:
        assert result["pattern_count"] >= 0, "Pattern count should be non-negative"


================================================================================
FILE: tests/integration/test_api_integration.py
================================================================================

```

```
import json
import time
from datetime import datetime, timedelta, timezone
from unittest.mock import patch

import pytest
from flask.testing import FlaskClient
from flask_socketio import SocketIOTestClient

from farfan_pipeline.api.api_server import (
    APIConfig,
    DataService,
    app,
    cache,
    cache_timestamps,
    generate_jwt_token,
    request_counts,
    socketio,
    verify_jwt_token,
)


@pytest.fixture
def client() -> FlaskClient:
    app.config["TESTING"] = True
    with app.test_client() as client:
        yield client


@pytest.fixture
def socketio_client() -> SocketIOTestClient:
    return socketio.test_client(app)


@pytest.fixture
def auth_token() -> str:
    return generate_jwt_token("test_client")


@pytest.fixture
def auth_headers(auth_token: str) -> dict[str, str]:
    return {"Authorization": f"Bearer {auth_token}"}


@pytest.fixture(autouse=True)
def clear_cache_and_rate_limits():
    cache.clear()
    cache_timestamps.clear()
    request_counts.clear()
    yield
    cache.clear()
    cache_timestamps.clear()
    request_counts.clear()


class TestHealthEndpoint:
```

```
4814:     def test_health_check_returns_200(self, client: FlaskClient):
4815:         response = client.get("/api/v1/health")
4816:         assert response.status_code == 200
4817:
4818:     def test_health_check_response_structure(self, client: FlaskClient):
4819:         response = client.get("/api/v1/health")
4820:         data = response.get_json()
4821:         assert "status" in data
4822:         assert "timestamp" in data
4823:         assert "version" in data
4824:         assert data["status"] == "healthy"
4825:
4826:     def test_health_check_timestamp_format(self, client: FlaskClient):
4827:         response = client.get("/api/v1/health")
4828:         data = response.get_json()
4829:         timestamp = data["timestamp"]
4830:         datetime.fromisoformat(timestamp)
4831:
4832:
4833: class TestAuthEndpoint:
4834:     def test_auth_token_missing_credentials(self, client: FlaskClient):
4835:         response = client.post(
4836:             "/api/v1/auth/token", data=json.dumps({}), content_type="application/json"
4837:         )
4838:         assert response.status_code == 400
4839:         data = response.get_json()
4840:         assert "error" in data
4841:         assert "Missing credentials" in data["error"]
4842:
4843:     def test_auth_token_missing_client_id(self, client: FlaskClient):
4844:         response = client.post(
4845:             "/api/v1/auth/token",
4846:             data=json.dumps({"client_secret": "secret"}),
4847:             content_type="application/json",
4848:         )
4849:         assert response.status_code == 400
4850:
4851:     def test_auth_token_missing_client_secret(self, client: FlaskClient):
4852:         response = client.post(
4853:             "/api/v1/auth/token",
4854:             data=json.dumps({"client_id": "test"}),
4855:             content_type="application/json",
4856:         )
4857:         assert response.status_code == 400
4858:
4859:     def test_auth_token_successful_generation(self, client: FlaskClient):
4860:         response = client.post(
4861:             "/api/v1/auth/token",
4862:             data=json.dumps(
4863:                 {"client_id": "test_client", "client_secret": "test_secret"}
4864:             ),
4865:             content_type="application/json",
4866:         )
4867:         assert response.status_code == 200
4868:         data = response.get_json()
4869:         assert "access_token" in data
```

```
4870:         assert "token_type" in data
4871:         assert "expires_in" in data
4872:         assert data["token_type"] == "Bearer"
4873:
4874:     def test_auth_token_is_valid_jwt(self, client: FlaskClient):
4875:         response = client.post(
4876:             "/api/v1/auth/token",
4877:             data=json.dumps(
4878:                 {"client_id": "test_client", "client_secret": "test_secret"}
4879:             ),
4880:             content_type="application/json",
4881:         )
4882:         data = response.get_json()
4883:         token = data["access_token"]
4884:         payload = verify_jwt_token(token)
4885:         assert payload is not None
4886:         assert payload["client_id"] == "test_client"
4887:
4888:     def test_auth_token_expiration(self):
4889:         token = generate_jwt_token("test_client")
4890:         payload = verify_jwt_token(token)
4891:         assert payload is not None
4892:         exp = datetime.fromtimestamp(payload["exp"])
4893:         iat = datetime.fromtimestamp(payload["iat"])
4894:         assert (exp - iat).total_seconds() == APIConfig.JWT_EXPIRATION_HOURS * 3600
4895:
4896:
4897: class TestPDETRegionsEndpoint:
4898:     def test_get_pdet_regions_returns_200(self, client: FlaskClient):
4899:         response = client.get("/api/v1/pdet/regions")
4900:         assert response.status_code == 200
4901:
4902:     def test_get_pdet_regions_response_structure(self, client: FlaskClient):
4903:         response = client.get("/api/v1/pdet/regions")
4904:         data = response.get_json()
4905:         assert "status" in data
4906:         assert "data" in data
4907:         assert "count" in data
4908:         assert "timestamp" in data
4909:         assert data["status"] == "success"
4910:
4911:     def test_get_pdet_regions_returns_16_regions(self, client: FlaskClient):
4912:         response = client.get("/api/v1/pdet/regions")
4913:         data = response.get_json()
4914:         assert data["count"] == 16
4915:         assert len(data["data"]) == 16
4916:
4917:     def test_get_pdet_regions_data_structure(self, client: FlaskClient):
4918:         response = client.get("/api/v1/pdet/regions")
4919:         data = response.get_json()
4920:         region = data["data"][0]
4921:         assert "id" in region
4922:         assert "name" in region
4923:         assert "coordinates" in region
4924:         assert "metadata" in region
4925:         assert "scores" in region
```

```
4926:         assert "connections" in region
4927:         assert "indicators" in region
4928:
4929:     def test_get_pdet_regions_scores_structure(self, client: FlaskClient):
4930:         response = client.get("/api/v1/pdet/regions")
4931:         data = response.get_json()
4932:         region = data["data"][0]
4933:         scores = region["scores"]
4934:         assert "overall" in scores
4935:         assert "governance" in scores
4936:         assert "social" in scores
4937:         assert "economic" in scores
4938:         assert "environmental" in scores
4939:         assert "lastUpdated" in scores
4940:
4941:     def test_get_pdet_regions_all_regions_have_ids(self, client: FlaskClient):
4942:         response = client.get("/api/v1/pdet/regions")
4943:         data = response.get_json()
4944:         region_ids = [r["id"] for r in data["data"]]
4945:         assert "alto-patia" in region_ids
4946:         assert "arauca" in region_ids
4947:         assert "choco" in region_ids
4948:         assert "uraba" in region_ids
4949:
4950:
4951: class TestPDETRegionDetailEndpoint:
4952:     def test_get_region_detail_returns_200(self, client: FlaskClient):
4953:         response = client.get("/api/v1/pdet/regions/alto-patia")
4954:         assert response.status_code == 200
4955:
4956:     def test_get_region_detail_not_found(self, client: FlaskClient):
4957:         response = client.get("/api/v1/pdet/regions/nonexistent")
4958:         assert response.status_code == 404
4959:         data = response.get_json()
4960:         assert "error" in data
4961:
4962:     def test_get_region_detail_response_structure(self, client: FlaskClient):
4963:         response = client.get("/api/v1/pdet/regions/alto-patia")
4964:         data = response.get_json()
4965:         assert "status" in data
4966:         assert "data" in data
4967:         assert "timestamp" in data
4968:         assert data["status"] == "success"
4969:
4970:     def test_get_region_detail_includes_detailed_analysis(self, client: FlaskClient):
4971:         response = client.get("/api/v1/pdet/regions/alto-patia")
4972:         data = response.get_json()
4973:         region = data["data"]
4974:         assert "detailed_analysis" in region
4975:         analysis = region["detailed_analysis"]
4976:         assert "cluster_breakdown" in analysis
4977:         assert "question_matrix" in analysis
4978:         assert "recommendations" in analysis
4979:         assert "evidence" in analysis
4980:
4981:     def test_get_region_detail_cluster_breakdown_structure(self, client: FlaskClient):
```

```
4982:         response = client.get("/api/v1/pdet/regions/alto-patia")
4983:         data = response.get_json()
4984:         clusters = data["data"]["detailed_analysis"]["cluster_breakdown"]
4985:         assert len(clusters) > 0
4986:         cluster = clusters[0]
4987:         assert "name" in cluster
4988:         assert "value" in cluster
4989:         assert "trend" in cluster
4990:
4991:     def test_get_region_detail_question_matrix_has_44_questions(
4992:         self, client: FlaskClient
4993:     ):
4994:         response = client.get("/api/v1/pdet/regions/alto-patia")
4995:         data = response.get_json()
4996:         questions = data["data"]["detailed_analysis"]["question_matrix"]
4997:         assert len(questions) == 44
4998:
4999:     def test_get_region_detail_all_valid_regions(self, client: FlaskClient):
5000:         region_ids = ["alto-patia", "arauca", "choco", "uraba"]
5001:         for region_id in region_ids:
5002:             response = client.get(f"/api/v1/pdet/regions/{region_id}")
5003:             assert response.status_code == 200
5004:
5005:
5006: class TestConstellationMapEndpoint:
5007:     def test_get_constellation_map_returns_200(self, client: FlaskClient):
5008:         response = client.get("/api/v1/constellation_map")
5009:         assert response.status_code == 200
5010:
5011:     def test_get_constellation_map_response_structure(self, client: FlaskClient):
5012:         response = client.get("/api/v1/constellation_map")
5013:         data = response.get_json()
5014:         assert "status" in data
5015:         assert "data" in data
5016:         assert "timestamp" in data
5017:         assert data["status"] == "success"
5018:
5019:     def test_get_constellation_map_has_nodes_and_links(self, client: FlaskClient):
5020:         response = client.get("/api/v1/constellation_map")
5021:         data = response.get_json()
5022:         constellation = data["data"]
5023:         assert "nodes" in constellation
5024:         assert "links" in constellation
5025:         assert len(constellation["nodes"]) > 0
5026:         assert len(constellation["links"]) > 0
5027:
5028:
5029: class TestMunicipalitiesEndpoint:
5030:     def test_get_municipality_returns_200(self, client: FlaskClient):
5031:         response = client.get("/api/v1/municipalities/test123")
5032:         assert response.status_code == 200
5033:
5034:     def test_get_municipality_response_structure(self, client: FlaskClient):
5035:         response = client.get("/api/v1/municipalities/test123")
5036:         data = response.get_json()
5037:         assert "status" in data
```

```
5038:         assert "data" in data
5039:         assert "timestamp" in data
5040:         assert data["status"] == "success"
5041:
5042:     def test_get_municipality_data_structure(self, client: FlaskClient):
5043:         response = client.get("/api/v1/municipalities/test123")
5044:         data = response.get_json()
5045:         municipality = data["data"]
5046:         assert "id" in municipality
5047:         assert "name" in municipality
5048:         assert "region_id" in municipality
5049:         assert "analysis" in municipality
5050:
5051:     def test_get_municipality_analysis_includes_radar(self, client: FlaskClient):
5052:         response = client.get("/api/v1/municipalities/test123")
5053:         data = response.get_json()
5054:         analysis = data["data"]["analysis"]
5055:         assert "radar" in analysis
5056:         radar = analysis["radar"]
5057:         assert "dimensions" in radar
5058:         assert "scores" in radar
5059:         assert len(radar["dimensions"]) == len(radar["scores"])
5060:
5061:     def test_get_municipality_analysis_includes_clusters(self, client: FlaskClient):
5062:         response = client.get("/api/v1/municipalities/test123")
5063:         data = response.get_json()
5064:         analysis = data["data"]["analysis"]
5065:         assert "clusters" in analysis
5066:         assert len(analysis["clusters"]) > 0
5067:
5068:     def test_get_municipality_analysis_includes_questions(self, client: FlaskClient):
5069:         response = client.get("/api/v1/municipalities/test123")
5070:         data = response.get_json()
5071:         analysis = data["data"]["analysis"]
5072:         assert "questions" in analysis
5073:         assert len(analysis["questions"]) == 44
5074:
5075:
5076: class TestEvidenceStreamEndpoint:
5077:     def test_get_evidence_stream_returns_200(self, client: FlaskClient):
5078:         response = client.get("/api/v1/evidence/stream")
5079:         assert response.status_code == 200
5080:
5081:     def test_get_evidence_stream_response_structure(self, client: FlaskClient):
5082:         response = client.get("/api/v1/evidence/stream")
5083:         data = response.get_json()
5084:         assert "status" in data
5085:         assert "data" in data
5086:         assert "count" in data
5087:         assert "timestamp" in data
5088:         assert data["status"] == "success"
5089:
5090:     def test_get_evidence_stream_has_evidence_items(self, client: FlaskClient):
5091:         response = client.get("/api/v1/evidence/stream")
5092:         data = response.get_json()
5093:         assert data["count"] > 0
```

```
5094:         assert len(data["data"]) > 0
5095:
5096:     def test_get_evidence_stream_item_structure(self, client: FlaskClient):
5097:         response = client.get("/api/v1/evidence/stream")
5098:         data = response.get_json()
5099:         item = data["data"][0]
5100:         assert "source" in item
5101:         assert "page" in item or "text" in item
5102:         assert "timestamp" in item
5103:
5104:
5105: class TestExportDashboardEndpoint:
5106:     def test_export_dashboard_json_format(self, client: FlaskClient):
5107:         response = client.post(
5108:             "/api/v1/export/dashboard",
5109:             data=json.dumps(
5110:                 {
5111:                     "format": "json",
5112:                     "regions": ["alto-patia", "arauca"],
5113:                     "include_evidence": True,
5114:                 }
5115:             ),
5116:             content_type="application/json",
5117:         )
5118:         assert response.status_code == 200
5119:         data = response.get_json()
5120:         assert "status" in data
5121:         assert "data" in data
5122:         assert data["status"] == "success"
5123:
5124:     def test_export_dashboard_response_structure(self, client: FlaskClient):
5125:         response = client.post(
5126:             "/api/v1/export/dashboard",
5127:             data=json.dumps(
5128:                 {"format": "json", "regions": ["alto-patia"], "include_evidence": False}
5129:             ),
5130:             content_type="application/json",
5131:         )
5132:         data = response.get_json()
5133:         export_data = data["data"]
5134:         assert "timestamp" in export_data
5135:         assert "regions" in export_data
5136:         assert "evidence" in export_data
5137:
5138:     def test_export_dashboard_includes_regions(self, client: FlaskClient):
5139:         response = client.post(
5140:             "/api/v1/export/dashboard",
5141:             data=json.dumps(
5142:                 {
5143:                     "format": "json",
5144:                     "regions": ["alto-patia", "arauca"],
5145:                     "include_evidence": False,
5146:                 }
5147:             ),
5148:             content_type="application/json",
5149:         )
```

```
        data = response.get_json()
        regions = data["data"]["regions"]
        assert len(regions) > 0

    def test_export_dashboard_unsupported_format(self, client: FlaskClient):
        response = client.post(
            "/api/v1/export/dashboard",
            data=json.dumps({"format": "pdf", "regions": ["alto-patia"]}),
            content_type="application/json",
        )
        assert response.status_code == 400
        data = response.get_json()
        assert "error" in data


class TestRecommendationMicroEndpoint:
    def test_micro_recommendations_missing_scores(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/micro",
            data=json.dumps({}),
            content_type="application/json",
        )
        assert response.status_code == 400 or response.status_code == 503

    def test_micro_recommendations_with_scores(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/micro",
            data=json.dumps(
                {
                    "scores": {"PA01-DIM01": 1.2, "PA02-DIM02": 1.5, "PA03-DIM01": 0.8},
                    "context": {},
                }
            ),
            content_type="application/json",
        )
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code == 200
        data = response.get_json()
        assert "status" in data
        assert "data" in data
        assert "timestamp" in data


class TestRecommendationMesoEndpoint:
    def test_meso_recommendations_missing_cluster_data(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/meso",
            data=json.dumps({}),
            content_type="application/json",
        )
        assert response.status_code == 400 or response.status_code == 503

    def test_meso_recommendations_with_cluster_data(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/meso",
```

```
            data=json.dumps(
                {
                    "cluster_data": {
                        "CL01": {"score": 72.0, "variance": 0.25, "weak_pa": "PA02"},
                        "CL02": {"score": 65.0, "variance": 0.30, "weak_pa": "PA05"},
                    },
                    "context": {},
                }
            ),
            content_type="application/json",
        )
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code == 200
        data = response.get_json()
        assert "status" in data
        assert "data" in data


class TestRecommendationMacroEndpoint:
    def test_macro_recommendations_missing_macro_data(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/macro",
            data=json.dumps({}),
            content_type="application/json",
        )
        assert response.status_code == 400 or response.status_code == 503

    def test_macro_recommendations_with_macro_data(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/macro",
            data=json.dumps(
                {
                    "macro_data": {
                        "macro_band": "SATISFACTORIO",
                        "clusters_below_target": ["CL02", "CL03"],
                        "variance_alert": "MODERADA",
                        "priority_micro_gaps": ["PA01-DIM05", "PA04-DIM04"],
                    },
                    "context": {},
                }
            ),
            content_type="application/json",
        )
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code == 200
        data = response.get_json()
        assert "status" in data
        assert "data" in data


class TestRecommendationAllEndpoint:
    def test_all_recommendations_complete_data(self, client: FlaskClient):
        response = client.post(
            "/api/v1/recommendations/all",
```

```
            data=json.dumps(
                {
                    "micro_scores": {"PA01-DIM01": 1.2, "PA02-DIM02": 1.5},
                    "cluster_data": {"CL01": {"score": 72.0, "variance": 0.25}},
                    "macro_data": {"macro_band": "SATISFACTORIO"},
                    "context": {},
                }
            ),
            content_type="application/json",
        )
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code == 200
        data = response.get_json()
        assert "status" in data
        assert "data" in data
        assert "summary" in data
        if data["status"] == "success":
            assert "MICRO" in data["data"]
            assert "MESO" in data["data"]
            assert "MACRO" in data["data"]


class TestRecommendationRulesInfoEndpoint:
    def test_rules_info_returns_structure(self, client: FlaskClient):
        response = client.get("/api/v1/recommendations/rules/info")
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code == 200
        data = response.get_json()
        assert "status" in data
        assert "data" in data


class TestRecommendationReloadEndpoint:
    def test_reload_rules_requires_auth(self, client: FlaskClient):
        response = client.post("/api/v1/recommendations/reload")
        assert response.status_code == 401

    def test_reload_rules_with_auth(
        self, client: FlaskClient, auth_headers: dict[str, str]
    ):
        response = client.post("/api/v1/recommendations/reload", headers=auth_headers)
        if response.status_code == 503:
            pytest.skip("Recommendation engine not available")
        assert response.status_code in [200, 401, 503]


class TestCachingValidation:
    def test_caching_on_pdet_regions(self, client: FlaskClient):
        response1 = client.get("/api/v1/pdet/regions")
        response2 = client.get("/api/v1/pdet/regions")
        assert response1.status_code == 200
        assert response2.status_code == 200
        assert response1.get_json() == response2.get_json()

```

```
    def test_cache_invalidation_after_ttl(self, client: FlaskClient):
        response1 = client.get("/api/v1/evidence/stream")
        time.sleep(0.1)
        response2 = client.get("/api/v1/evidence/stream")
        assert response1.status_code == 200
        assert response2.status_code == 200

    def test_different_endpoints_have_separate_cache(self, client: FlaskClient):
        response1 = client.get("/api/v1/pdet/regions")
        response2 = client.get("/api/v1/evidence/stream")
        assert response1.status_code == 200
        assert response2.status_code == 200
        assert response1.get_json() != response2.get_json()

    def test_cache_key_includes_query_params(self, client: FlaskClient):
        response1 = client.get("/api/v1/pdet/regions")
        response2 = client.get("/api/v1/pdet/regions?test=1")
        assert response1.status_code == 200
        assert response2.status_code == 200


class TestRateLimiting:
    def test_rate_limit_not_exceeded_for_normal_requests(self, client: FlaskClient):
        for _ in range(5):
            response = client.get("/api/v1/health")
            assert response.status_code == 200

    def test_rate_limit_tracks_per_endpoint(self, client: FlaskClient):
        for _ in range(3):
            response1 = client.get("/api/v1/health")
            response2 = client.get("/api/v1/pdet/regions")
            assert response1.status_code == 200
            assert response2.status_code == 200

    @pytest.mark.skipif(
        not APIConfig.RATE_LIMIT_ENABLED, reason="Rate limiting disabled"
    )
    def test_rate_limit_exceeded_returns_429(self, client: FlaskClient):
        original_limit = APIConfig.RATE_LIMIT_REQUESTS
        APIConfig.RATE_LIMIT_REQUESTS = 5

        responses = []
        for _ in range(10):
            response = client.get("/api/v1/health")
            responses.append(response.status_code)

        APIConfig.RATE_LIMIT_REQUESTS = original_limit

        if 429 in responses:
            assert True
        else:
            pytest.skip("Rate limit not triggered in test")


class TestWebSocketSSE:
    def test_websocket_connection(self, socketio_client: SocketIOTestClient):
```

```
5374:         assert socketio_client.is_connected()
5375:
5376:     def test_websocket_connect_response(self, socketio_client: SocketIOTestClient):
5377:         received = socketio_client.get_received()
5378:         connection_responses = [
5379:             msg for msg in received if msg.get("name") == "connection_response"
5380:         ]
5381:         assert len(connection_responses) > 0
5382:         assert connection_responses[0]["args"][0]["status"] == "connected"
5383:
5384:     def test_websocket_subscribe_region(self, socketio_client: SocketIOTestClient):
5385:         socketio_client.emit("subscribe_region", {"region_id": "alto-patia"})
5386:         received = socketio_client.get_received()
5387:         region_updates = [msg for msg in received if msg.get("name") == "region_update"]
5388:         assert len(region_updates) > 0
5389:
5390:     def test_websocket_disconnect(self):
5391:         client = socketio.test_client(app)
5392:         assert client.is_connected()
5393:         client.disconnect()
5394:         assert not client.is_connected()
5395:
5396:
5397: class TestAuthenticationValidation:
5398:     def test_require_auth_decorator_blocks_unauthenticated(self, client: FlaskClient):
5399:         response = client.post("/api/v1/recommendations/reload")
5400:         assert response.status_code == 401
5401:         data = response.get_json()
5402:         assert "error" in data
5403:
5404:     def test_require_auth_decorator_accepts_valid_token(
5405:         self, client: FlaskClient, auth_headers: dict[str, str]
5406:     ):
5407:         response = client.post("/api/v1/recommendations/reload", headers=auth_headers)
5408:         assert response.status_code != 401
5409:
5410:     def test_require_auth_decorator_rejects_invalid_token(self, client: FlaskClient):
5411:         response = client.post(
5412:             "/api/v1/recommendations/reload",
5413:             headers={"Authorization": "Bearer invalid_token_xyz"},
5414:         )
5415:         assert response.status_code == 401
5416:
5417:     def test_require_auth_decorator_rejects_malformed_header(self, client: FlaskClient):
5418:         response = client.post(
5419:             "/api/v1/recommendations/reload",
5420:             headers={"Authorization": "invalid_format"},
5421:         )
5422:         assert response.status_code == 401
5423:
5424:     def test_expired_token_rejected(self):
5425:         with patch("farfan_pipeline.api.api_server.datetime") as mock_datetime:
5426:             past_time = datetime.now(timezone.utc) - timedelta(hours=25)
5427:             mock_datetime.now.return_value = past_time
5428:             token = generate_jwt_token("test_client")
5429:
```

```
            mock_datetime.now.return_value = datetime.now(timezone.utc)
            payload = verify_jwt_token(token)
            assert payload is None


class TestErrorHandling:
    def test_404_for_invalid_endpoint(self, client: FlaskClient):
        response = client.get("/api/v1/nonexistent")
        assert response.status_code == 404

    def test_405_for_wrong_method(self, client: FlaskClient):
        response = client.get("/api/v1/auth/token")
        assert response.status_code == 405

    def test_400_for_invalid_json(self, client: FlaskClient):
        response = client.post(
            "/api/v1/export/dashboard",
            data="invalid json{",
            content_type="application/json",
        )
        assert response.status_code in [400, 500]

    def test_region_not_found_returns_404(self, client: FlaskClient):
        response = client.get("/api/v1/pdet/regions/invalid-region-id-xyz")
        assert response.status_code == 404


class TestEndpointCompleteness:
    def test_all_endpoints_documented(self, client: FlaskClient):
        endpoints = [
            ("/", "GET"),
            ("/api/v1/health", "GET"),
            ("/api/v1/auth/token", "POST"),
            ("/api/v1/constellation_map", "GET"),
            ("/api/v1/pdet/regions", "GET"),
            ("/api/v1/pdet/regions/alto-patia", "GET"),
            ("/api/v1/municipalities/test123", "GET"),
            ("/api/v1/evidence/stream", "GET"),
            ("/api/v1/export/dashboard", "POST"),
            ("/api/v1/recommendations/micro", "POST"),
            ("/api/v1/recommendations/meso", "POST"),
            ("/api/v1/recommendations/macro", "POST"),
            ("/api/v1/recommendations/all", "POST"),
            ("/api/v1/recommendations/rules/info", "GET"),
        ]

        for endpoint, method in endpoints:
            if method == "GET":
                response = client.get(endpoint)
            else:
                response = client.post(
                    endpoint, data=json.dumps({}), content_type="application/json"
                )
            assert response.status_code in [200, 400, 401, 404, 503]

    def test_12_main_rest_endpoints_covered(self):
```

```
5486:         expected_endpoints = {
5487:             "health",
5488:             "auth_token",
5489:             "constellation_map",
5490:             "pdet_regions",
5491:             "region_detail",
5492:             "municipality",
5493:             "evidence_stream",
5494:             "export_dashboard",
5495:             "micro_recommendations",
5496:             "meso_recommendations",
5497:             "macro_recommendations",
5498:             "all_recommendations",
5499:         }
5500:         assert len(expected_endpoints) == 12
5501:
5502:
5503: class TestCORS:
5504:     def test_cors_headers_present(self, client: FlaskClient):
5505:         response = client.get("/api/v1/health")
5506:         assert response.status_code == 200
5507:
5508:     def test_options_request_for_cors(self, client: FlaskClient):
5509:         response = client.options("/api/v1/health")
5510:         assert response.status_code in [200, 204]
5511:
5512:
5513: class TestDataServiceIntegration:
5514:     def test_data_service_returns_16_regions(self):
5515:         service = DataService()
5516:         regions = service.get_pdet_regions()
5517:         assert len(regions) == 16
5518:
5519:     def test_data_service_region_detail(self):
5520:         service = DataService()
5521:         region = service.get_region_detail("alto-patia")
5522:         assert region is not None
5523:         assert region["id"] == "alto-patia"
5524:
5525:     def test_data_service_evidence_stream(self):
5526:         service = DataService()
5527:         evidence = service.get_evidence_stream()
5528:         assert len(evidence) > 0
5529:
5530:     def test_data_service_constellation_map(self):
5531:         service = DataService()
5532:         constellation = service.get_constellation_map_data()
5533:         assert "nodes" in constellation
5534:         assert "links" in constellation
5535:
5536:
5537:
5538: ================================================================================
5539: FILE: tests/integration/test_irrigation_synchronizer.py
5540: ================================================================================
5541:
```

```
5542: """Integration tests for IrrigationSynchronizer.
5543:
5544: Tests the full synchronization flow from questionnaire â\206\222 chunks â\206\222 tasks â\206\222 execution plan.
5545: """
5546:
5547: import json
5548: import logging
5549: import re
5550: import sys
5551: from pathlib import Path
5552: from typing import Any
5553:
5554: import pytest
5555:
5556: project_root = Path(__file__).parent.parent.parent
5557: sys.path.insert(0, str(project_root / "src"))
5558:
5559: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
5560:     ExecutionPlan,
5561:     IrrigationSynchronizer,
5562:     Task,
5563: )
5564:
5565:
5566: @pytest.fixture
5567: def questionnaire_data() -> dict[str, Any]:
5568:     """Load real questionnaire_monolith.json."""
5569:     questionnaire_path = Path("system/config/questionnaire/questionnaire_monolith.json")
5570:
5571:     if not questionnaire_path.exists():
5572:         pytest.skip(f"Questionnaire not found at {questionnaire_path}")
5573:
5574:     with open(questionnaire_path) as f:
5575:         return json.load(f)
5576:
5577:
5578: @pytest.fixture
5579: def mock_document_chunks() -> list[dict[str, Any]]:
5580:     """Generate 60 mock document chunks."""
5581:     return [
5582:         {
5583:             "chunk_id": f"chunk_{i:04d}",
5584:             "text": f"Sample text for chunk {i}",
5585:             "start_char": i * 1000,
5586:             "end_char": (i + 1) * 1000,
5587:         }
5588:         for i in range(60)
5589:     ]
5590:
5591:
5592: def test_synchronizer_initialization(questionnaire_data, mock_document_chunks):
5593:     """Test synchronizer initializes with correlation_id and logging."""
5594:     synchronizer = IrrigationSynchronizer(
5595:         questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5596:     )
5597:
```

```
5598:     assert synchronizer.correlation_id is not None
5599:     assert len(synchronizer.correlation_id) == 36
5600:     assert synchronizer.question_count > 0
5601:     assert synchronizer.chunk_count == 60
5602:
5603:
5604: def test_build_execution_plan_basic(questionnaire_data, mock_document_chunks):
5605:     """Test build_execution_plan generates valid ExecutionPlan."""
5606:     synchronizer = IrrigationSynchronizer(
5607:         questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5608:     )
5609:
5610:     plan = synchronizer.build_execution_plan()
5611:
5612:     assert isinstance(plan, ExecutionPlan)
5613:     assert plan.plan_id.startswith("plan_")
5614:     assert len(plan.tasks) > 0
5615:     assert plan.chunk_count == 60
5616:     assert plan.question_count > 0
5617:     assert len(plan.integrity_hash) in [64, 128]
5618:     assert plan.correlation_id == synchronizer.correlation_id
5619:
5620:
5621: def test_execution_plan_determinism(questionnaire_data, mock_document_chunks):
5622:     """Test execution plan generation is deterministic."""
5623:     sync1 = IrrigationSynchronizer(
5624:         questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5625:     )
5626:     plan1 = sync1.build_execution_plan()
5627:
5628:     sync2 = IrrigationSynchronizer(
5629:         questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5630:     )
5631:     plan2 = sync2.build_execution_plan()
5632:
5633:     assert plan1.plan_id == plan2.plan_id
5634:     assert plan1.integrity_hash == plan2.integrity_hash
5635:     assert len(plan1.tasks) == len(plan2.tasks)
5636:
5637:
5638: def test_task_structure(questionnaire_data, mock_document_chunks):
5639:     """Test individual task structure."""
5640:     synchronizer = IrrigationSynchronizer(
5641:         questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5642:     )
5643:
5644:     plan = synchronizer.build_execution_plan()
5645:
5646:     task = plan.tasks[0]
5647:     assert isinstance(task, Task)
5648:     assert task.task_id
5649:     assert task.dimension in ["D1", "D2", "D3", "D4", "D5", "D6"]
5650:     assert task.question_id
5651:     assert task.policy_area in [f"PA{i:02d}" for i in range(1, 11)]
5652:     assert task.chunk_id
5653:     assert task.chunk_index >= 0
```

```
5654:     assert task.question_text
5655:
5656:
5657: def test_build_execution_plan_empty_chunks():
5658:     """Test synchronizer raises ValueError for empty chunks."""
5659:     synchronizer = IrrigationSynchronizer(
5660:         questionnaire={"blocks": {}}, document_chunks=[]
5661:     )
5662:
5663:     with pytest.raises(ValueError, match="No document chunks provided"):
5664:         synchronizer.build_execution_plan()
5665:
5666:
5667: def test_build_execution_plan_empty_questionnaire(mock_document_chunks):
5668:     """Test synchronizer raises ValueError for empty questionnaire."""
5669:     synchronizer = IrrigationSynchronizer(
5670:         questionnaire={"blocks": {}}, document_chunks=mock_document_chunks
5671:     )
5672:
5673:     with pytest.raises(ValueError, match="No questions found in questionnaire"):
5674:         synchronizer.build_execution_plan()
5675:
5676:
5677: def test_logging_includes_correlation_id(
5678:     questionnaire_data, mock_document_chunks, caplog
5679: ):
5680:     """Test all log entries include the same correlation_id."""
5681:     with caplog.at_level(logging.INFO):
5682:         synchronizer = IrrigationSynchronizer(
5683:             questionnaire=questionnaire_data, document_chunks=mock_document_chunks
5684:         )
5685:
5686:         correlation_id = synchronizer.correlation_id
5687:
5688:         plan = synchronizer.build_execution_plan()
5689:
5690:     correlation_ids = []
5691:     for record in caplog.records:
5692:         try:
5693:             log_data = json.loads(record.message)
5694:             if "correlation_id" in log_data:
5695:                 correlation_ids.append(log_data["correlation_id"])
5696:         except (json.JSONDecodeError, AttributeError):
5697:             pass
5698:
5699:     assert len(correlation_ids) >= 2
5700:     assert all(cid == correlation_id for cid in correlation_ids)
5701:
5702:
5703: def test_end_to_end_synchronization_with_real_questionnaire(questionnaire_data):
5704:     """Test complete synchronization flow with real questionnaire and 60 chunks.
5705:
5706:     This is the canonical end-to-end test that verifies:
5707:     - 60 document chunks are processed
5708:     - Execution plan is generated with 300 tasks
5709:     - plan_id is deterministic
```

```
    - integrity_hash is reproducible
    """
    chunks = [
        {
            "chunk_id": f"chunk_{i:04d}",
            "text": f"Policy section {i} with substantive content about development goals",
            "start_char": i * 2000,
            "end_char": (i + 1) * 2000,
            "chunk_type": "semantic",
        }
        for i in range(60)
    ]

    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire_data, document_chunks=chunks
    )

    plan = synchronizer.build_execution_plan()

    assert plan.chunk_count == 60, f"Expected 60 chunks, got {plan.chunk_count}"

    assert len(plan.tasks) > 0, "Execution plan should contain tasks"

    assert plan.plan_id is not None and plan.plan_id.startswith("plan_")

    assert len(plan.integrity_hash) in [64, 128]

    assert plan.question_count > 0

    integrity_hash_original = plan.integrity_hash

    synchronizer2 = IrrigationSynchronizer(
        questionnaire=questionnaire_data, document_chunks=chunks
    )
    plan2 = synchronizer2.build_execution_plan()

    assert (
        plan2.integrity_hash == integrity_hash_original
    ), "Integrity hash should be deterministic"

    assert plan2.plan_id == plan.plan_id, "plan_id should be deterministic"


def test_execution_plan_serialization(questionnaire_data, mock_document_chunks):
    """Test ExecutionPlan can be serialized to dict and JSON."""
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire_data, document_chunks=mock_document_chunks
    )

    plan = synchronizer.build_execution_plan()

    plan_dict = plan.to_dict()

    assert isinstance(plan_dict, dict)
    assert "plan_id" in plan_dict
    assert "tasks" in plan_dict
```

```
    assert "chunk_count" in plan_dict
    assert "question_count" in plan_dict
    assert "integrity_hash" in plan_dict
    assert "correlation_id" in plan_dict

    json_str = json.dumps(plan_dict)
    assert len(json_str) > 0

    deserialized = json.loads(json_str)
    assert deserialized["plan_id"] == plan.plan_id
    assert deserialized["integrity_hash"] == plan.integrity_hash


def test_prometheus_metrics_integration(questionnaire_data, mock_document_chunks):
    """Test that Prometheus metrics are recorded (if available)."""
    try:
        from prometheus_client import REGISTRY

        synchronizer = IrrigationSynchronizer(
            questionnaire=questionnaire_data, document_chunks=mock_document_chunks
        )

        plan = synchronizer.build_execution_plan()

        metrics = REGISTRY.collect()
        metric_names = [m.name for m in metrics]

        assert any("synchronization" in name for name in metric_names)

    except ImportError:
        pytest.skip("prometheus_client not available")


def test_correlation_id_uniqueness():
    """Test each synchronizer instance gets unique correlation_id."""
    sync1 = IrrigationSynchronizer(
        questionnaire={"blocks": {"D1_Q01": {"question": "Test"}}},
        document_chunks=[{"chunk_id": "c1", "text": "test"}],
    )

    sync2 = IrrigationSynchronizer(
        questionnaire={"blocks": {"D1_Q01": {"question": "Test"}}},
        document_chunks=[{"chunk_id": "c1", "text": "test"}],
    )

    assert sync1.correlation_id != sync2.correlation_id


def test_task_count_calculation(questionnaire_data, mock_document_chunks):
    """Test task count matches expected formula: questions Ã\227 policy_areas Ã\227 chunks."""
    synchronizer = IrrigationSynchronizer(
        questionnaire=questionnaire_data, document_chunks=mock_document_chunks
    )

    plan = synchronizer.build_execution_plan()

```

```
5822:     expected_task_count = plan.question_count * 10 * 60
5823:
5824:     assert (
5825:         len(plan.tasks) == expected_task_count
5826:     ), f"Expected {expected_task_count} tasks, got {len(plan.tasks)}"
5827:
5828:
5829: if __name__ == "__main__":
5830:     pytest.main([__file__, "-v", "--tb=short"])
5831:
5832:
5833:
5834: ================================================================================
5835: FILE: tests/integration/test_pipeline_e2e_deterministic.py
5836: ================================================================================
5837:
5838: """End-to-End Deterministic Pipeline Integration Tests
5839:
5840: Validates Phase 0 through Phase 9 execution with:
5841: - Fixed seed (seed=42) determinism
5842: - BLAKE3 phase hash stability
5843: - verification_manifest.json structure validation
5844: - Failure propagation testing (Phase N failure to ABORT)
5845: - provenance_completeness=1.0 verification
5846: - 150-page test document processing
5847:
5848: Author: F.A.R.F.A.N Test Team
5849: Date: 2025-01-19
5850: """
5851:
5852: import json
5853: import tempfile
5854: from pathlib import Path
5855: from typing import Any
5856:
5857: import blake3
5858: import pytest
5859:
5860:
5861: @pytest.fixture
5862: def test_artifacts_dir():
5863:     with tempfile.TemporaryDirectory() as tmpdir:
5864:         yield Path(tmpdir)
5865:
5866:
5867: @pytest.fixture
5868: def generate_150_page_test_pdf(test_artifacts_dir):
5869:     try:
5870:         import fitz
5871:     except ImportError:
5872:         pytest.skip("PyMuPDF not available")
5873:
5874:     pdf_path = test_artifacts_dir / "test_plan_150pages.pdf"
5875:     doc = fitz.open()
5876:
5877:     for page_num in range(150):
```

```
5878:         page = doc.new_page(width=595, height=842)
5879:         page.insert_text(
5880:             (50, 50), f"Plan de Desarrollo - PÃ¡gina {page_num + 1}", fontsize=14, fontname="helv"
5881:         )
5882:
5883:         content_type = page_num % 6
5884:         if content_type == 0:
5885:             text = f"DiagnÃ³stico {page_num + 1}: AnÃ¡lisis de brechas."
5886:         elif content_type == 1:
5887:             text = f"Actividad {page_num + 1}: Programa de capacitaciÃ³n."
5888:         elif content_type == 2:
5889:             text = f"Indicador {page_num + 1}: Beneficiarios atendidos."
5890:         elif content_type == 3:
5891:             text = f"Recursos {page_num + 1}: Presupuesto $250,000 USD."
5892:         elif content_type == 4:
5893:             text = f"Cronograma {page_num + 1}: Inicio enero 2025."
5894:         else:
5895:             text = f"Entidades {page_num + 1}: Municipalidad distrital."
5896:
5897:         rect = fitz.Rect(50, 80, 545, 792)
5898:         page.insert_textbox(rect, text, fontsize=11, fontname="helv")
5899:         page.insert_text((50, 820), f"PÃ¡gina {page_num + 1}/150", fontsize=8, fontname="helv")
5900:
5901:     doc.save(pdf_path)
5902:     doc.close()
5903:     return pdf_path
5904:
5905:
5906: @pytest.fixture
5907: def questionnaire_file():
5908:     from farfan_core.config.paths import QUESTIONNAIRE_FILE
5909:
5910:     if not QUESTIONNAIRE_FILE.exists():
5911:         pytest.skip(f"Questionnaire not found: {QUESTIONNAIRE_FILE}")
5912:     return QUESTIONNAIRE_FILE
5913:
5914:
5915: def compute_phase_hash_blake3(phase_data: dict[str, Any]) -> str:
5916:     json_str = json.dumps(phase_data, sort_keys=True, separators=(",", ":"))
5917:     return blake3.blake3(json_str.encode("utf-8")).hexdigest()
5918:
5919:
5920: class TestPipelineE2EDeterministic:
5921:
5922:     @pytest.mark.asyncio
5923:     async def test_phase_0_to_9_execution_with_fixed_seed(
5924:         self, generate_150_page_test_pdf, questionnaire_file, test_artifacts_dir
5925:     ):
5926:         from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator
5927:
5928:         orchestrator = PhaseOrchestrator()
5929:         result = await orchestrator.run_pipeline(
5930:             pdf_path=generate_150_page_test_pdf,
5931:             run_id="test_e2e_seed42",
5932:             questionnaire_path=questionnaire_file,
5933:             artifacts_dir=test_artifacts_dir,
```

```
        )

        assert result.success is True, f"Pipeline failed: {result.errors}"
        assert result.phases_completed >= 4
        assert result.phases_failed == 0
        assert result.canonical_input is not None
        assert result.canonical_input.pdf_page_count == 150
        assert result.canonical_input.validation_passed is True
        assert len(result.canonical_input.pdf_sha256) == 64
        assert len(result.canonical_input.questionnaire_sha256) == 64
        assert result.canon_policy_package is not None
        assert hasattr(result.canon_policy_package, "chunk_graph")
        assert result.preprocessed_document is not None
        assert result.preprocessed_document.processing_mode == "chunked"
        assert len(result.preprocessed_document.sentences) > 0
        assert result.phase2_result is not None
        questions = result.phase2_result.get("questions", [])
        assert len(questions) > 0
        assert result.manifest is not None
        assert "phases" in result.manifest

    @pytest.mark.asyncio
    async def test_blake3_phase_hash_stability(
        self, generate_150_page_test_pdf, questionnaire_file, test_artifacts_dir
    ):
        from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator

        orchestrator = PhaseOrchestrator()
        result1 = await orchestrator.run_pipeline(
            pdf_path=generate_150_page_test_pdf,
            run_id="test_hash_run1",
            questionnaire_path=questionnaire_file,
            artifacts_dir=test_artifacts_dir / "run1",
        )
        result2 = await orchestrator.run_pipeline(
            pdf_path=generate_150_page_test_pdf,
            run_id="test_hash_run2",
            questionnaire_path=questionnaire_file,
            artifacts_dir=test_artifacts_dir / "run2",
        )

        assert result1.success is True
        assert result2.success is True

        phase0_data1 = {
            "pdf_sha256": result1.canonical_input.pdf_sha256,
            "pdf_page_count": result1.canonical_input.pdf_page_count,
            "questionnaire_sha256": result1.canonical_input.questionnaire_sha256,
        }
        phase0_data2 = {
            "pdf_sha256": result2.canonical_input.pdf_sha256,
            "pdf_page_count": result2.canonical_input.pdf_page_count,
            "questionnaire_sha256": result2.canonical_input.questionnaire_sha256,
        }

        hash1 = compute_phase_hash_blake3(phase0_data1)
```

```
5990:         hash2 = compute_phase_hash_blake3(phase0_data2)
5991:         assert hash1 == hash2
5992:         assert len(hash1) == 64
5993:
5994:     @pytest.mark.asyncio
5995:     async def test_verification_manifest_structure(
5996:         self, generate_150_page_test_pdf, questionnaire_file, test_artifacts_dir
5997:     ):
5998:         from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator
5999:
6000:         orchestrator = PhaseOrchestrator()
6001:         result = await orchestrator.run_pipeline(
6002:             pdf_path=generate_150_page_test_pdf,
6003:             run_id="test_manifest_structure",
6004:             questionnaire_path=questionnaire_file,
6005:             artifacts_dir=test_artifacts_dir,
6006:         )
6007:
6008:         assert result.success is True
6009:         manifest = result.manifest
6010:         assert "phases" in manifest
6011:         assert "total_phases" in manifest
6012:         assert "successful_phases" in manifest
6013:         assert "failed_phases" in manifest
6014:
6015:         phases = manifest["phases"]
6016:         assert "phase0_input_validation" in phases
6017:         assert "phase1_spc_ingestion" in phases
6018:         assert "phase1_to_phase2_adapter" in phases
6019:
6020:         phase0 = phases["phase0_input_validation"]
6021:         assert "status" in phase0
6022:         assert "started_at" in phase0
6023:         assert "finished_at" in phase0
6024:         assert "duration_ms" in phase0
6025:         assert "input_contract" in phase0
6026:         assert "output_contract" in phase0
6027:         assert "invariants_checked" in phase0
6028:         assert phase0["status"] == "success"
6029:         assert phase0["input_contract"]["validation_passed"] is True
6030:         assert phase0["output_contract"]["validation_passed"] is True
6031:
6032:         phase1 = phases["phase1_spc_ingestion"]
6033:         assert phase1["status"] == "success"
6034:         assert phase1["duration_ms"] > 0
6035:
6036:         adapter = phases["phase1_to_phase2_adapter"]
6037:         assert adapter["status"] == "success"
6038:         assert adapter["duration_ms"] > 0
6039:
6040:         manifest_file = test_artifacts_dir / "phase_manifest.json"
6041:         assert manifest_file.exists()
6042:
6043:         with open(manifest_file) as f:
6044:             saved_manifest = json.load(f)
6045:         assert "phases" in saved_manifest
```

```
6046:
6047:     @pytest.mark.asyncio
6048:     async def test_phase_failure_propagation_abort(self, test_artifacts_dir, questionnaire_file):
6049:         from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator
6050:
6051:         orchestrator = PhaseOrchestrator()
6052:         fake_pdf_path = test_artifacts_dir / "nonexistent.pdf"
6053:         result = await orchestrator.run_pipeline(
6054:             pdf_path=fake_pdf_path,
6055:             run_id="test_phase_failure",
6056:             questionnaire_path=questionnaire_file,
6057:             artifacts_dir=test_artifacts_dir,
6058:         )
6059:
6060:         assert result.success is False
6061:         assert len(result.errors) > 0
6062:         assert result.phases_failed > 0
6063:         manifest = result.manifest
6064:         assert manifest is not None
6065:
6066:         phases = manifest.get("phases", {})
6067:         if "phase0_input_validation" in phases:
6068:             phase0 = phases["phase0_input_validation"]
6069:             assert phase0["status"] == "failed"
6070:
6071:     @pytest.mark.asyncio
6072:     async def test_provenance_completeness(
6073:         self, generate_150_page_test_pdf, questionnaire_file, test_artifacts_dir
6074:     ):
6075:         from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator
6076:
6077:         orchestrator = PhaseOrchestrator()
6078:         result = await orchestrator.run_pipeline(
6079:             pdf_path=generate_150_page_test_pdf,
6080:             run_id="test_provenance",
6081:             questionnaire_path=questionnaire_file,
6082:             artifacts_dir=test_artifacts_dir,
6083:         )
6084:
6085:         assert result.success is True
6086:         assert result.preprocessed_document is not None
6087:
6088:         chunks = result.preprocessed_document.chunks
6089:         assert len(chunks) > 0
6090:
6091:         chunks_with_provenance = 0
6092:         for chunk in chunks:
6093:             if chunk.provenance is not None:
6094:                 chunks_with_provenance += 1
6095:                 assert chunk.provenance.page_number > 0
6096:                 assert chunk.provenance.page_number <= 150
6097:
6098:         provenance_completeness = chunks_with_provenance / len(chunks)
6099:         assert provenance_completeness == 1.0, (
6100:             f"Expected provenance_completeness=1.0, " f"got {provenance_completeness}"
6101:         )
```

```
6102:
6103:     @pytest.mark.asyncio
6104:     async def test_phase_hash_collection_all_phases(
6105:         self, generate_150_page_test_pdf, questionnaire_file, test_artifacts_dir
6106:     ):
6107:         from farfan_core.core.phases.phase_orchestrator import PhaseOrchestrator
6108:
6109:         orchestrator = PhaseOrchestrator()
6110:         result = await orchestrator.run_pipeline(
6111:             pdf_path=generate_150_page_test_pdf,
6112:             run_id="test_phase_hashes",
6113:             questionnaire_path=questionnaire_file,
6114:             artifacts_dir=test_artifacts_dir,
6115:         )
6116:
6117:         assert result.success is True
6118:         manifest = result.manifest
6119:         phases = manifest["phases"]
6120:
6121:         phase_hashes = {}
6122:         for phase_name, phase_data in phases.items():
6123:             if phase_data["status"] == "success":
6124:                 hash_data = {
6125:                     "phase_name": phase_name,
6126:                     "duration_ms": phase_data["duration_ms"],
6127:                     "started_at": phase_data["started_at"],
6128:                 }
6129:                 phase_hash = compute_phase_hash_blake3(hash_data)
6130:                 phase_hashes[phase_name] = phase_hash
6131:                 assert len(phase_hash) == 64
6132:
6133:         assert len(phase_hashes) >= 3
6134:         assert "phase0_input_validation" in phase_hashes
6135:         assert "phase1_spc_ingestion" in phase_hashes
6136:         assert "phase1_to_phase2_adapter" in phase_hashes
6137:
6138:
6139:
6140: ================================================================================
6141: FILE: tests/integration/test_routing_contract.py
6142: ================================================================================
6143:
6144: """
6145: Comprehensive test suite for ExecutionMap contract validation.
6146:
6147: These tests serve as CI-blocking guarantees of routing correctness.
6148: Any failure represents a critical regression and must prevent code merges.
6149: """
6150:
6151: import json
6152:
6153: import pytest
6154:
6155: from farfan_pipeline.core.orchestrator.chunk_router import (
6156:     ChunkRoute,
6157:     ChunkRouter,
```

```
    compute_execution_map_hash,
    deserialize_execution_map,
    serialize_execution_map,
)
from farfan_pipeline.core.types import ChunkData


@pytest.fixture
def chunk_router():
    return ChunkRouter()


class TestContractValidation:
    """Verify ExecutionMap object validity and contract compliance."""

    def test_generate_execution_map_returns_valid_structure(self, chunk_router):
        chunks = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=1.0,
            )
        ]

        execution_map = chunk_router.generate_execution_map(chunks)

        assert isinstance(execution_map, dict)
        assert len(execution_map) == 1
        assert 1 in execution_map
        assert isinstance(execution_map[1], ChunkRoute)

    def test_chunk_route_has_required_fields(self, chunk_router):
        chunk = ChunkData(
            id=42,
            text="Test chunk",
            chunk_type="activity",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
        )

        route = chunk_router.route_chunk(chunk)

        assert hasattr(route, "chunk_id")
        assert hasattr(route, "chunk_type")
        assert hasattr(route, "executor_class")
        assert hasattr(route, "methods")
        assert hasattr(route, "skip_reason")
        assert route.chunk_id == 42
```

```
6214:         assert route.chunk_type == "activity"
6215:
6216:     def test_execution_map_with_mixed_chunk_types(self, chunk_router):
6217:         chunks = [
6218:             ChunkData(
6219:                 id=1,
6220:                 text="Diagnostic",
6221:                 chunk_type="diagnostic",
6222:                 sentences=[],
6223:                 tables=[],
6224:                 start_pos=0,
6225:                 end_pos=10,
6226:                 confidence=1.0,
6227:             ),
6228:             ChunkData(
6229:                 id=2,
6230:                 text="Activity",
6231:                 chunk_type="activity",
6232:                 sentences=[],
6233:                 tables=[],
6234:                 start_pos=11,
6235:                 end_pos=20,
6236:                 confidence=1.0,
6237:             ),
6238:             ChunkData(
6239:                 id=3,
6240:                 text="Indicator",
6241:                 chunk_type="indicator",
6242:                 sentences=[],
6243:                 tables=[],
6244:                 start_pos=21,
6245:                 end_pos=30,
6246:                 confidence=1.0,
6247:             ),
6248:         ]
6249:
6250:         execution_map = chunk_router.generate_execution_map(chunks)
6251:
6252:         assert len(execution_map) == 3
6253:         assert all(isinstance(route, ChunkRoute) for route in execution_map.values())
6254:         assert execution_map[1].chunk_type == "diagnostic"
6255:         assert execution_map[2].chunk_type == "activity"
6256:         assert execution_map[3].chunk_type == "indicator"
6257:
6258:
6259: class TestSerializationContract:
6260:     """Verify serialization/deserialization is lossless and correct."""
6261:
6262:     def test_serialize_execution_map_produces_valid_json(self, chunk_router):
6263:         chunks = [
6264:             ChunkData(
6265:                 id=1,
6266:                 text="Test",
6267:                 chunk_type="diagnostic",
6268:                 sentences=[],
6269:                 tables=[],
```

```
                start_pos=0,
                end_pos=4,
                confidence=1.0,
            )
        ]

        execution_map = chunk_router.generate_execution_map(chunks)
        serialized = serialize_execution_map(execution_map)

        parsed = json.loads(serialized)
        assert "version" in parsed
        assert "routes" in parsed
        assert isinstance(parsed["routes"], dict)

    def test_serialize_deserialize_roundtrip_preserves_data(self, chunk_router):
        chunks = [
            ChunkData(
                id=10,
                text="Diagnostic chunk",
                chunk_type="diagnostic",
                sentences=[1, 2],
                tables=[],
                start_pos=0,
                end_pos=16,
                confidence=0.95,
                policy_area_id="PA05",
                dimension_id="DIM01",
            ),
            ChunkData(
                id=20,
                text="Activity chunk",
                chunk_type="activity",
                sentences=[3],
                tables=[1],
                start_pos=17,
                end_pos=31,
                confidence=0.88,
                policy_area_id="PA03",
                dimension_id="DIM02",
            ),
        ]

        original_map = chunk_router.generate_execution_map(chunks)
        serialized = serialize_execution_map(original_map)
        restored_map = deserialize_execution_map(serialized)

        assert len(restored_map) == len(original_map)
        assert set(restored_map.keys()) == set(original_map.keys())

        for chunk_id in original_map:
            orig_route = original_map[chunk_id]
            rest_route = restored_map[chunk_id]

            assert orig_route.chunk_id == rest_route.chunk_id
            assert orig_route.chunk_type == rest_route.chunk_type
            assert orig_route.executor_class == rest_route.executor_class
```

```
            assert orig_route.methods == rest_route.methods
            assert orig_route.skip_reason == rest_route.skip_reason

    def test_deserialize_rejects_invalid_json(self):
        invalid_json = "{ invalid json }"
        with pytest.raises(ValueError, match="Invalid JSON format"):
            deserialize_execution_map(invalid_json)

    def test_deserialize_rejects_wrong_version(self):
        wrong_version = json.dumps({"version": "v999", "routes": {}})
        with pytest.raises(ValueError, match="Invalid or unsupported version"):
            deserialize_execution_map(wrong_version)

    def test_deserialize_rejects_missing_routes(self):
        missing_routes = json.dumps({"version": "v1"})
        with pytest.raises(ValueError, match="Missing 'routes' key"):
            deserialize_execution_map(missing_routes)

    def test_deserialize_rejects_invalid_chunk_id(self):
        invalid_id = json.dumps(
            {
                "version": "v1",
                "routes": {
                    "not_an_int": {
                        "chunk_id": 1,
                        "chunk_type": "diagnostic",
                        "executor_class": "D1Q1",
                        "methods": [],
                        "skip_reason": None,
                    }
                },
            }
        )
        with pytest.raises(ValueError, match="Invalid chunk_id"):
            deserialize_execution_map(invalid_id)

    def test_serialize_produces_deterministic_output(self, chunk_router):
        chunks = [
            ChunkData(
                id=5,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=1.0,
            )
        ]

        execution_map = chunk_router.generate_execution_map(chunks)

        serialized1 = serialize_execution_map(execution_map)
        serialized2 = serialize_execution_map(execution_map)

        assert serialized1 == serialized2
```

```
6382:
6383:
6384: class TestDeterminismContract:
6385:     """Verify ExecutionMap generation is completely deterministic."""
6386:
6387:     def test_multiple_runs_produce_identical_maps(self, chunk_router):
6388:         chunks = [
6389:             ChunkData(
6390:                 id=1,
6391:                 text="Test diagnostic",
6392:                 chunk_type="diagnostic",
6393:                 sentences=[],
6394:                 tables=[],
6395:                 start_pos=0,
6396:                 end_pos=15,
6397:                 confidence=0.9,
6398:             ),
6399:             ChunkData(
6400:                 id=2,
6401:                 text="Test activity",
6402:                 chunk_type="activity",
6403:                 sentences=[],
6404:                 tables=[],
6405:                 start_pos=16,
6406:                 end_pos=29,
6407:                 confidence=0.85,
6408:             ),
6409:         ]
6410:
6411:         maps = [chunk_router.generate_execution_map(chunks) for _ in range(10)]
6412:
6413:         first_map = maps[0]
6414:         for subsequent_map in maps[1:]:
6415:             assert list(first_map.keys()) == list(subsequent_map.keys())
6416:             for chunk_id in first_map:
6417:                 assert first_map[chunk_id] == subsequent_map[chunk_id]
6418:
6419:     def test_map_hash_is_deterministic(self, chunk_router):
6420:         chunks = [
6421:             ChunkData(
6422:                 id=7,
6423:                 text="Test",
6424:                 chunk_type="resource",
6425:                 sentences=[],
6426:                 tables=[],
6427:                 start_pos=0,
6428:                 end_pos=4,
6429:                 confidence=1.0,
6430:             )
6431:         ]
6432:
6433:         execution_map = chunk_router.generate_execution_map(chunks)
6434:
6435:         hashes = [compute_execution_map_hash(execution_map) for _ in range(5)]
6436:
6437:         assert len(set(hashes)) == 1
```

```
6438:
6439:     def test_input_order_does_not_affect_determinism(self, chunk_router):
6440:         chunk1 = ChunkData(
6441:             id=100,
6442:             text="First",
6443:             chunk_type="diagnostic",
6444:             sentences=[],
6445:             tables=[],
6446:             start_pos=0,
6447:             end_pos=5,
6448:             confidence=1.0,
6449:         )
6450:         chunk2 = ChunkData(
6451:             id=200,
6452:             text="Second",
6453:             chunk_type="activity",
6454:             sentences=[],
6455:             tables=[],
6456:             start_pos=6,
6457:             end_pos=12,
6458:             confidence=1.0,
6459:         )
6460:
6461:         map_order1 = chunk_router.generate_execution_map([chunk1, chunk2])
6462:         map_order2 = chunk_router.generate_execution_map([chunk2, chunk1])
6463:
6464:         hash1 = compute_execution_map_hash(map_order1)
6465:         hash2 = compute_execution_map_hash(map_order2)
6466:
6467:         assert hash1 == hash2
6468:
6469:     def test_determinism_with_policy_area_and_dimension(self, chunk_router):
6470:         chunks = [
6471:             ChunkData(
6472:                 id=1,
6473:                 text="Chunk 1",
6474:                 chunk_type="diagnostic",
6475:                 sentences=[],
6476:                 tables=[],
6477:                 start_pos=0,
6478:                 end_pos=7,
6479:                 confidence=1.0,
6480:                 policy_area_id="PA05",
6481:                 dimension_id="DIM02",
6482:             ),
6483:             ChunkData(
6484:                 id=2,
6485:                 text="Chunk 2",
6486:                 chunk_type="activity",
6487:                 sentences=[],
6488:                 tables=[],
6489:                 start_pos=8,
6490:                 end_pos=15,
6491:                 confidence=1.0,
6492:                 policy_area_id="PA03",
6493:                 dimension_id="DIM04",
```

```
            ),
        ]

        hashes = [
            compute_execution_map_hash(chunk_router.generate_execution_map(chunks))
            for _ in range(10)
        ]

        assert len(set(hashes)) == 1


class TestCorrectnessContract:
    """Verify routing correctness for specific policy_area and dimension combinations."""

    @pytest.mark.parametrize(
        "chunk_type,expected_executor",
        [
            ("diagnostic", "D1Q1"),
            ("activity", "D2Q1"),
            ("indicator", "D3Q1"),
            ("resource", "D1Q3"),
            ("temporal", "D1Q5"),
            ("entity", "D2Q3"),
        ],
    )
    def test_chunk_type_maps_to_correct_primary_executor(
        self, chunk_router, chunk_type, expected_executor
    ):
        chunk = ChunkData(
            id=1,
            text="Test",
            chunk_type=chunk_type,
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=4,
            confidence=1.0,
        )

        route = chunk_router.route_chunk(chunk)

        assert route.executor_class == expected_executor
        assert route.skip_reason is None

    def test_pa05_dim02_routes_correctly(self, chunk_router):
        chunk = ChunkData(
            id=1,
            text="PA05 + DIM02 specific content",
            chunk_type="activity",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=29,
            confidence=0.95,
            policy_area_id="PA05",
            dimension_id="DIM02",
```

```
        )

        route = chunk_router.route_chunk(chunk)

        assert route.executor_class == "D2Q1"
        assert route.skip_reason is None

    def test_unknown_chunk_type_produces_skip_reason(self, chunk_router):
        chunk = ChunkData(
            id=999,
            text="Unknown type",
            chunk_type="unknown_type",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=12,
            confidence=0.5,
        )

        route = chunk_router.route_chunk(chunk)

        assert route.executor_class == ""
        assert route.skip_reason is not None
        assert "No executor mapping" in route.skip_reason

    def test_multiple_pa_dim_combinations_route_exclusively(self, chunk_router):
        chunks = [
            ChunkData(
                id=1,
                text="PA01 DIM01",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=1.0,
                policy_area_id="PA01",
                dimension_id="DIM01",
            ),
            ChunkData(
                id=2,
                text="PA05 DIM02",
                chunk_type="activity",
                sentences=[],
                tables=[],
                start_pos=11,
                end_pos=21,
                confidence=1.0,
                policy_area_id="PA05",
                dimension_id="DIM02",
            ),
            ChunkData(
                id=3,
                text="PA10 DIM06",
                chunk_type="indicator",
                sentences=[],
```

```
                tables=[],
                start_pos=22,
                end_pos=32,
                confidence=1.0,
                policy_area_id="PA10",
                dimension_id="DIM06",
            ),
        ]

        execution_map = chunk_router.generate_execution_map(chunks)

        assert execution_map[1].executor_class == "D1Q1"
        assert execution_map[2].executor_class == "D2Q1"
        assert execution_map[3].executor_class == "D3Q1"

        executors = {route.executor_class for route in execution_map.values()}
        assert len(executors) == 3

    def test_routing_table_completeness(self, chunk_router):
        valid_chunk_types = [
            "diagnostic",
            "activity",
            "indicator",
            "resource",
            "temporal",
            "entity",
        ]

        for chunk_type in valid_chunk_types:
            chunk = ChunkData(
                id=1,
                text=f"Test {chunk_type}",
                chunk_type=chunk_type,
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=1.0,
            )

            route = chunk_router.route_chunk(chunk)

            assert route.executor_class != ""
            assert route.skip_reason is None


class TestEdgeCasesContract:
    """Verify router behavior with malformed or incomplete chunk metadata."""

    def test_chunk_with_none_policy_area(self, chunk_router):
        chunk = ChunkData(
            id=1,
            text="Test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
```

```
            start_pos=0,
            end_pos=4,
            confidence=1.0,
            policy_area_id=None,
            dimension_id="DIM01",
        )

        route = chunk_router.route_chunk(chunk)

        assert isinstance(route, ChunkRoute)
        assert route.executor_class == "D1Q1"

    def test_chunk_with_none_dimension(self, chunk_router):
        chunk = ChunkData(
            id=1,
            text="Test",
            chunk_type="activity",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=4,
            confidence=1.0,
            policy_area_id="PA05",
            dimension_id=None,
        )

        route = chunk_router.route_chunk(chunk)

        assert isinstance(route, ChunkRoute)
        assert route.executor_class == "D2Q1"

    def test_empty_chunk_list(self, chunk_router):
        execution_map = chunk_router.generate_execution_map([])

        assert execution_map == {}

    def test_chunk_with_zero_confidence(self, chunk_router):
        chunk = ChunkData(
            id=1,
            text="Low confidence",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=14,
            confidence=0.0,
        )

        route = chunk_router.route_chunk(chunk)

        assert isinstance(route, ChunkRoute)
        assert route.executor_class == "D1Q1"

    def test_chunk_with_negative_positions(self, chunk_router):
        chunk = ChunkData(
            id=1,
```

```
            text="Negative positions",
            chunk_type="activity",
            sentences=[],
            tables=[],
            start_pos=-10,
            end_pos=-1,
            confidence=1.0,
        )

        route = chunk_router.route_chunk(chunk)

        assert isinstance(route, ChunkRoute)

    def test_chunk_with_empty_text(self, chunk_router):
        chunk = ChunkData(
            id=1,
            text="",
            chunk_type="indicator",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=0,
            confidence=1.0,
        )

        route = chunk_router.route_chunk(chunk)

        assert isinstance(route, ChunkRoute)
        assert route.executor_class == "D3Q1"

    def test_large_number_of_chunks(self, chunk_router):
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic" if i % 2 == 0 else "activity",
                sentences=[],
                tables=[],
                start_pos=i * 10,
                end_pos=i * 10 + 10,
                confidence=1.0,
            )
            for i in range(1000)
        ]

        execution_map = chunk_router.generate_execution_map(chunks)

        assert len(execution_map) == 1000
        assert all(isinstance(route, ChunkRoute) for route in execution_map.values())

    def test_deserialize_with_extra_fields_is_tolerant(self):
        serialized_with_extra = json.dumps(
            {
                "version": "v1",
                "extra_field": "should_be_ignored",
                "routes": {
```

```
                    "1": {
                        "chunk_id": 1,
                        "chunk_type": "diagnostic",
                        "executor_class": "D1Q1",
                        "methods": [],
                        "skip_reason": None,
                        "extra_route_field": "also_ignored",
                    }
                },
            }
        )

        execution_map = deserialize_execution_map(serialized_with_extra)

        assert len(execution_map) == 1
        assert execution_map[1].chunk_id == 1

    def test_malformed_chunk_route_in_deserialization(self):
        malformed = json.dumps(
            {
                "version": "v1",
                "routes": {
                    "1": {
                        "chunk_id": 1,
                    }
                },
            }
        )

        with pytest.raises(ValueError, match="Invalid ChunkRoute data"):
            deserialize_execution_map(malformed)


class TestCIBlockingValidation:
    """
    Critical tests that MUST pass for CI to succeed.
    These tests validate the core routing contract guarantees.
    """

    def test_ci_blocking_routing_determinism(self, chunk_router):
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type=["diagnostic", "activity", "indicator"][i % 3],
                sentences=[],
                tables=[],
                start_pos=i * 10,
                end_pos=i * 10 + 10,
                confidence=0.9,
                policy_area_id=f"PA0{(i % 9) + 1}",
                dimension_id=f"DIM0{(i % 6) + 1}",
            )
            for i in range(30)
        ]

```

```
6830:         maps = [chunk_router.generate_execution_map(chunks) for _ in range(5)]
6831:
6832:         first_hash = compute_execution_map_hash(maps[0])
6833:         for subsequent_map in maps[1:]:
6834:             assert compute_execution_map_hash(subsequent_map) == first_hash
6835:
6836:     def test_ci_blocking_serialization_roundtrip(self, chunk_router):
6837:         chunks = [
6838:             ChunkData(
6839:                 id=i,
6840:                 text=f"Test chunk {i}",
6841:                 chunk_type=["diagnostic", "activity", "indicator", "resource"][i % 4],
6842:                 sentences=[i],
6843:                 tables=[],
6844:                 start_pos=i * 20,
6845:                 end_pos=i * 20 + 15,
6846:                 confidence=0.85 + (i * 0.01),
6847:                 policy_area_id=f"PA0{(i % 9) + 1}",
6848:                 dimension_id=f"DIM0{(i % 6) + 1}",
6849:             )
6850:             for i in range(20)
6851:         ]
6852:
6853:         original_map = chunk_router.generate_execution_map(chunks)
6854:         serialized = serialize_execution_map(original_map)
6855:         restored_map = deserialize_execution_map(serialized)
6856:
6857:         original_hash = compute_execution_map_hash(original_map)
6858:         restored_hash = compute_execution_map_hash(restored_map)
6859:
6860:         assert original_hash == restored_hash
6861:
6862:     def test_ci_blocking_routing_correctness(self, chunk_router):
6863:         test_cases = [
6864:             ("diagnostic", "D1Q1"),
6865:             ("activity", "D2Q1"),
6866:             ("indicator", "D3Q1"),
6867:             ("resource", "D1Q3"),
6868:             ("temporal", "D1Q5"),
6869:             ("entity", "D2Q3"),
6870:         ]
6871:
6872:         for chunk_type, expected_executor in test_cases:
6873:             chunk = ChunkData(
6874:                 id=1,
6875:                 text=f"Test {chunk_type}",
6876:                 chunk_type=chunk_type,
6877:                 sentences=[],
6878:                 tables=[],
6879:                 start_pos=0,
6880:                 end_pos=10,
6881:                 confidence=1.0,
6882:             )
6883:
6884:             route = chunk_router.route_chunk(chunk)
6885:
```

```
            assert (
                route.executor_class == expected_executor
            ), f"Chunk type '{chunk_type}' must route to '{expected_executor}', got '{route.executor_class}'"


================================================================================
FILE: tests/integration/test_signal_intelligence_integration.py
================================================================================

"""
Integration Test: Signal Intelligence Layer with Real Questionnaire
====================================================================

This test validates the complete signal flow using REAL data from the
questionnaire monolith, not mocks or stubs.

Test Coverage:
1. Load questionnaire via canonical interface
2. Create signal registry (modern flow)
3. Apply intelligence layer enhancements
4. Validate semantic expansion
5. Validate context filtering
6. Validate contract validation
7. Validate evidence extraction
8. End-to-end analysis pipeline

Author: F.A.R.F.A.N Pipeline
Date: 2025-12-02
Status: Severe Integration Test (NO MOCKS)
"""

import pytest
from pathlib import Path

# Canonical questionnaire access
from farfan_pipeline.core.orchestrator.questionnaire import (
    load_questionnaire,
    CanonicalQuestionnaire
)

# Modern signal registry
from farfan_pipeline.core.orchestrator.signal_registry import (
    QuestionnaireSignalRegistry,
    create_signal_registry
)

# Intelligence layer
from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
    create_enriched_signal_pack,
    analyze_with_intelligence_layer,
    EnrichedSignalPack,
    generate_precision_improvement_report
)

# Individual components
```

```
6942: from farfan_pipeline.core.orchestrator.signal_semantic_expander import (
6943:     expand_pattern_semantically,
6944:     expand_all_patterns
6945: )
6946: from farfan_pipeline.core.orchestrator.signal_context_scoper import (
6947:     filter_patterns_by_context,
6948:     create_document_context
6949: )
6950: from farfan_pipeline.core.orchestrator.signal_contract_validator import (
6951:     validate_with_contract,
6952:     execute_failure_contract
6953: )
6954: from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
6955:     extract_structured_evidence
6956: )
6957:
6958:
6959: class TestSignalIntelligenceIntegration:
6960:     """Integration tests for signal intelligence layer."""
6961:
6962:     @pytest.fixture(scope="class")
6963:     def canonical_questionnaire(self) -> CanonicalQuestionnaire:
6964:         """Load real questionnaire via canonical interface."""
6965:         return load_questionnaire()
6966:
6967:     @pytest.fixture(scope="class")
6968:     def signal_registry(self, canonical_questionnaire) -> QuestionnaireSignalRegistry:
6969:         """Create signal registry from questionnaire."""
6970:         return QuestionnaireSignalRegistry.from_questionnaire(canonical_questionnaire)
6971:
6972:     def test_01_canonical_questionnaire_loads(self, canonical_questionnaire):
6973:         """Test 1: Canonical questionnaire loads successfully."""
6974:         assert canonical_questionnaire is not None
6975:         assert canonical_questionnaire.data is not None
6976:
6977:         # Check structure
6978:         blocks = canonical_questionnaire.data.get('blocks', {})
6979:         assert 'micro_questions' in blocks
6980:
6981:         micro_questions = blocks['micro_questions']
6982:         assert len(micro_questions) > 0
6983:
6984:         print(f"\nâ\234\223 Loaded {len(micro_questions)} micro questions")
6985:
6986:     def test_02_signal_registry_initializes(self, signal_registry):
6987:         """Test 2: Signal registry initializes from questionnaire."""
6988:         assert signal_registry is not None
6989:         assert signal_registry.questionnaire is not None
6990:
6991:         # Check that signals can be retrieved
6992:         assert len(signal_registry.questionnaire.data['blocks']['micro_questions']) > 0
6993:
6994:         print(f"\nâ\234\223 Signal registry initialized")
6995:
6996:     def test_03_patterns_have_metadata(self, canonical_questionnaire):
6997:         """Test 3: Patterns contain rich metadata (not just strings)."""
```

```
6998:         blocks = canonical_questionnaire.data['blocks']
6999:         micro_questions = blocks['micro_questions']
7000:
7001:         # Check first question
7002:         mq = micro_questions[0]
7003:         patterns = mq.get('patterns', [])
7004:
7005:         assert len(patterns) > 0, "No patterns found in first question"
7006:
7007:         # Check metadata presence
7008:         sample_pattern = patterns[0]
7009:         assert 'pattern' in sample_pattern
7010:         assert 'confidence_weight' in sample_pattern
7011:         assert 'category' in sample_pattern
7012:         assert 'context_scope' in sample_pattern
7013:
7014:         print(f"\nâ\234\223 Pattern metadata present: {list(sample_pattern.keys())}")
7015:
7016:     def test_04_semantic_expansion_with_real_patterns(self, canonical_questionnaire):
7017:         """Test 4: Semantic expansion works with real patterns."""
7018:         blocks = canonical_questionnaire.data['blocks']
7019:         micro_questions = blocks['micro_questions']
7020:
7021:         # Find patterns with semantic_expansion
7022:         patterns_with_expansion = []
7023:         for mq in micro_questions:
7024:             for p in mq.get('patterns', []):
7025:                 if p.get('semantic_expansion'):
7026:                     patterns_with_expansion.append(p)
7027:
7028:         print(f"\nâ\234\223 Found {len(patterns_with_expansion)} patterns with semantic_expansion")
7029:
7030:         if patterns_with_expansion:
7031:             # Test expansion
7032:             sample = patterns_with_expansion[0]
7033:             variants = expand_pattern_semantically(sample)
7034:
7035:             assert len(variants) >= 1  # At least original
7036:             print(f"  Original pattern: {sample.get('pattern')}")
7037:             print(f"  Semantic expansion: {sample.get('semantic_expansion')}")
7038:             print(f"  Generated variants: {len(variants)}")
7039:
7040:             for v in variants[1:3]:  # Show first 2 variants
7041:                 print(f"    â\206\222 {v.get('id')}: {v.get('pattern')}")
7042:
7043:     def test_05_context_filtering_with_real_patterns(self, canonical_questionnaire):
7044:         """Test 5: Context filtering works with real patterns."""
7045:         blocks = canonical_questionnaire.data['blocks']
7046:         mq = blocks['micro_questions'][0]
7047:         patterns = mq.get('patterns', [])
7048:
7049:         # Test context filtering
7050:         context_budget = create_document_context(section='budget', chapter=3)
7051:         context_indicators = create_document_context(section='indicators', chapter=5)
7052:
7053:         filtered_budget, stats_budget = filter_patterns_by_context(patterns, context_budget)
```

```
        filtered_indicators, stats_indicators = filter_patterns_by_context(patterns, context_indicators)

        print(f"\nâ\234\223 Context filtering results:")
        print(f"  Budget context: {stats_budget['passed']}/{stats_budget['total_patterns']} patterns")
        print(f"  Indicators context: {stats_indicators['passed']}/{stats_indicators['total_patterns']} patterns")

        # Both should have some patterns (context_scope allows global patterns)
        assert stats_budget['passed'] > 0
        assert stats_indicators['passed'] > 0

    def test_06_failure_contract_validation(self, canonical_questionnaire):
        """Test 6: Failure contract validation works."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        failure_contract = mq.get('failure_contract', {})
        print(f"\nâ\234\223 Failure contract: {failure_contract}")

        if failure_contract:
            # Test with missing data (should fail)
            result_fail = {'completeness': 0.3, 'missing_elements': ['required_field']}
            validation_fail = execute_failure_contract(result_fail, failure_contract)

            print(f"  Failed validation: {validation_fail.status}")
            print(f"  Error code: {validation_fail.error_code}")

            # Test with complete data (should pass)
            result_pass = {'completeness': 1.0, 'missing_elements': []}
            validation_pass = execute_failure_contract(result_pass, failure_contract)

            print(f"  Passed validation: {validation_pass.status}")

    def test_07_evidence_extraction_with_real_elements(self, canonical_questionnaire):
        """Test 7: Evidence extraction with real expected_elements."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        expected_elements = mq.get('expected_elements', [])
        print(f"\nâ\234\223 Expected elements: {expected_elements}")

        # Simulate document text
        sample_text = """
        LÃ-nea de base: 8.5% de tasa de desempleo.
        Meta: reducir a 6% para 2027.
        Responsable: SecretarÃ-a de Desarrollo EconÃ³mico.
        Presupuesto: COP 1,500 millones.
        """

        # Extract evidence
        signal_node = {
            'expected_elements': [
                'baseline_indicator',
                'target_value',
                'timeline',
                'responsible_entity',
                'budget_amount'
```

```
            ],
            'patterns': mq.get('patterns', []),
            'validations': mq.get('validations', {})
        }

        evidence_result = extract_structured_evidence(sample_text, signal_node)

        print(f"  Completeness: {evidence_result.completeness:.2f}")
        print(f"  Extracted: {len(evidence_result.evidence)} elements")
        print(f"  Missing: {evidence_result.missing_elements}")

        for key, val in evidence_result.evidence.items():
            print(f"    - {key}: {val.get('value')} (confidence: {val.get('confidence', 0):.2f})")

        # Should extract at least some elements
        assert len(evidence_result.evidence) > 0

    def test_08_enriched_signal_pack_creation(self, signal_registry, canonical_questionnaire):
        """Test 8: Enriched signal pack wraps modern registry correctly."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        # Create a mock signal pack (in real usage, this comes from registry)
        class MockSignalPack:
            def __init__(self, patterns):
                self.patterns = patterns

        base_pack = MockSignalPack(mq.get('patterns', []))

        # Create enriched pack
        enriched = create_enriched_signal_pack(base_pack, enable_semantic_expansion=False)

        assert enriched is not None
        assert enriched.base_pack is base_pack
        assert len(enriched.patterns) >= len(base_pack.patterns)

        print(f"\nâ\234\223 Enriched signal pack created")
        print(f"  Base patterns: {len(base_pack.patterns)}")
        print(f"  Enriched patterns: {len(enriched.patterns)}")

    def test_09_enriched_pack_context_filtering(self, canonical_questionnaire):
        """Test 9: Enriched pack provides context-filtered patterns with stats."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        class MockSignalPack:
            def __init__(self, patterns):
                self.patterns = patterns

        base_pack = MockSignalPack(mq.get('patterns', []))
        enriched = create_enriched_signal_pack(base_pack, enable_semantic_expansion=False)

        # Get patterns for specific context
        context = create_document_context(section='budget', chapter=3)
        filtered_patterns, stats = enriched.get_patterns_for_context(context)

```

```
        print(f"\nâ\234\223 Context-filtered patterns: {len(filtered_patterns)}")
        print(f"  Total patterns: {stats['total_patterns']}")
        print(f"  Passed: {stats['passed']}")
        print(f"  Filter rate: {stats['filter_rate']:.1%}")
        print(f"  Precision improvement: {stats.get('precision_improvement', 0):.1%}")
        print(f"  False positive reduction: {stats.get('false_positive_reduction', 0):.1%}")
        print(f"  Integration validated: {stats.get('integration_validated', False)}")

        assert len(filtered_patterns) > 0
        assert stats['integration_validated'] is True

    def test_10_end_to_end_analysis_pipeline(self, canonical_questionnaire):
        """Test 10: Complete end-to-end analysis with intelligence layer."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        # Prepare signal node
        signal_node = {
            'expected_elements': [
                'baseline_indicator',
                'target_value',
                'timeline'
            ],
            'patterns': mq.get('patterns', []),
            'validations': mq.get('validations', {}),
            'failure_contract': mq.get('failure_contract', {})
        }

        # Sample document
        document_text = """
        El diagnÃ³stico de gÃ©nero presenta los siguientes datos:
        LÃnea de base aÃ±o 2023: 8.5% de mujeres en cargos directivos.
        Meta establecida: alcanzar 15% para el aÃ±o 2027.
        SegÃºn datos del DANE, la tasa ha permanecido estable.
        """

        # Document context
        context = create_document_context(
            section='indicators',
            chapter=2,
            page=15
        )

        # Run complete analysis
        result = analyze_with_intelligence_layer(
            text=document_text,
            signal_node=signal_node,
            document_context=context
        )

        print(f"\nâ\234\223 End-to-end analysis complete")
        print(f"  Evidence count: {len(result['evidence'])}")
        print(f"  Completeness: {result['completeness']:.2f}")
        print(f"  Validation status: {result['validation']['status']}")
        print(f"  Refactorings applied: {result['metadata']['refactorings_applied']}")

```

```
7222:         # Validate structure
7223:         assert 'evidence' in result
7224:         assert 'completeness' in result
7225:         assert 'validation' in result
7226:         assert 'metadata' in result
7227:
7228:         # Check metadata
7229:         assert result['metadata']['intelligence_layer_enabled'] is True
7230:         assert len(result['metadata']['refactorings_applied']) == 4
7231:
7232:     def test_11_pattern_expansion_multiplier(self, canonical_questionnaire):
7233:         """Test 11: Verify semantic expansion provides 2x+ multiplier."""
7234:         blocks = canonical_questionnaire.data['blocks']
7235:
7236:         # Collect all patterns
7237:         all_patterns = []
7238:         for mq in blocks['micro_questions'][:10]:  # Test first 10 questions
7239:             all_patterns.extend(mq.get('patterns', []))
7240:
7241:         original_count = len(all_patterns)
7242:
7243:         # Expand patterns
7244:         expanded = expand_all_patterns(all_patterns, enable_logging=False)
7245:         expanded_count = len(expanded)
7246:
7247:         multiplier = expanded_count / original_count if original_count > 0 else 1.0
7248:
7249:         print(f"\nâ\234\223 Pattern expansion multiplier test:")
7250:         print(f"  Original patterns: {original_count}")
7251:         print(f"  Expanded patterns: {expanded_count}")
7252:         print(f"  Multiplier: {multiplier:.2f}x")
7253:
7254:         # Should have at least original patterns
7255:         assert expanded_count >= original_count
7256:
7257:     def test_12_metadata_preservation_through_pipeline(self, canonical_questionnaire):
7258:         """Test 12: Metadata is preserved through the entire pipeline."""
7259:         blocks = canonical_questionnaire.data['blocks']
7260:         mq = blocks['micro_questions'][0]
7261:         patterns = mq.get('patterns', [])
7262:
7263:         if not patterns:
7264:             pytest.skip("No patterns available")
7265:
7266:         original_pattern = patterns[0]
7267:
7268:         # Verify original metadata
7269:         assert 'confidence_weight' in original_pattern
7270:         assert 'category' in original_pattern
7271:         assert 'context_scope' in original_pattern
7272:
7273:         original_confidence = original_pattern['confidence_weight']
7274:         original_category = original_pattern['category']
7275:
7276:         # Expand pattern
7277:         variants = expand_pattern_semantically(original_pattern)
```

```
7278:
7279:         # Check metadata preservation in variants
7280:         for variant in variants:
7281:             if variant.get('is_variant'):
7282:                 # Variants should inherit metadata
7283:                 assert variant.get('confidence_weight') == original_confidence
7284:                 assert variant.get('category') == original_category
7285:                 assert 'variant_of' in variant
7286:                 print(f"\nâ\234\223 Variant {variant.get('id')} inherits metadata:")
7287:                 print(f"  confidence_weight: {variant.get('confidence_weight')}")
7288:                 print(f"  category: {variant.get('category')}")
7289:
7290:
7291:     def test_13_precision_improvement_validation_with_real_patterns(self, canonical_questionnaire):
7292:         """Test 13: Validate 60% precision improvement target with real patterns."""
7293:         blocks = canonical_questionnaire.data['blocks']
7294:
7295:         # Test multiple policy areas to validate consistent precision improvement
7296:         precision_measurements = []
7297:
7298:         for mq_idx, mq in enumerate(blocks['micro_questions'][:5]):
7299:             patterns = mq.get('patterns', [])
7300:             if not patterns:
7301:                 continue
7302:
7303:             class MockSignalPack:
7304:                 def __init__(self, patterns):
7305:                     self.patterns = patterns
7306:
7307:             base_pack = MockSignalPack(patterns)
7308:             enriched = create_enriched_signal_pack(base_pack, enable_semantic_expansion=False)
7309:
7310:             # Test with multiple contexts
7311:             contexts = [
7312:                 create_document_context(section='budget', chapter=1),
7313:                 create_document_context(section='indicators', chapter=2),
7314:                 create_document_context(section='diagnosis', chapter=3),
7315:             ]
7316:
7317:             for ctx in contexts:
7318:                 filtered_patterns, stats = enriched.get_patterns_for_context(ctx)
7319:
7320:                 measurement = {
7321:                     'mq_index': mq_idx,
7322:                     'context': ctx.get('section'),
7323:                     'filter_rate': stats['filter_rate'],
7324:                     'false_positive_reduction': stats['false_positive_reduction'],
7325:                     'precision_improvement': stats['precision_improvement'],
7326:                     'estimated_final_precision': stats['estimated_final_precision'],
7327:                     'integration_validated': stats['integration_validated']
7328:                 }
7329:                 precision_measurements.append(measurement)
7330:
7331:         # Verify measurements collected
7332:         assert len(precision_measurements) > 0
7333:
```

```
        # Generate comprehensive report
        report = generate_precision_improvement_report(precision_measurements)

        print(f"\n{report['summary']}")

        # Assertions
        assert report['validated_count'] > 0, "No measurements validated integration"
        assert report['avg_false_positive_reduction'] >= 0.0, "Average FP reduction should be non-negative"
        assert report['max_false_positive_reduction'] <= 0.60, "Max FP reduction should not exceed 60% cap"
        assert report['avg_final_precision'] >= 0.40, "Final precision should improve from baseline"

        # Show sample measurements
        print("\n  Sample measurements:")
        for m in precision_measurements[:3]:
            print(f"    MQ{m['mq_index']} {m['context']}: "
                  f"filter={m['filter_rate']:.1%}, "
                  f"FP_reduction={m['false_positive_reduction']:.1%}, "
                  f"precision={m['estimated_final_precision']:.1%}")


class TestSignalFlowCompliance:
    """Test compliance with access control and architectural rules."""

    def test_01_questionnaire_access_via_canonical_only(self):
        """Test: Questionnaire must be accessed via canonical interface."""
        # This test ensures we're using the canonical loader
        questionnaire = load_questionnaire()

        assert isinstance(questionnaire, CanonicalQuestionnaire)
        assert questionnaire.data is not None

        print("\nâ\234\223 Questionnaire accessed via canonical interface")

    def test_02_signal_registry_uses_canonical_questionnaire(self):
        """Test: Signal registry must use CanonicalQuestionnaire."""
        questionnaire = load_questionnaire()
        registry = QuestionnaireSignalRegistry.from_questionnaire(questionnaire)

        assert registry.questionnaire is questionnaire

        print("\nâ\234\223 Signal registry uses canonical questionnaire")

    def test_03_intelligence_layer_backward_compatible(self, canonical_questionnaire):
        """Test: Intelligence layer works with both legacy and modern."""
        blocks = canonical_questionnaire.data['blocks']
        mq = blocks['micro_questions'][0]

        class LegacySignalPack:
            def __init__(self):
                self.patterns = [p.get('pattern', '') for p in mq.get('patterns', [])]

        class ModernSignalPack:
            def __init__(self):
                self.patterns = mq.get('patterns', [])

        # Test with legacy (strings only)
```

```
7390:         legacy_pack = LegacySignalPack()
7391:         try:
7392:             enriched_legacy = create_enriched_signal_pack(legacy_pack, enable_semantic_expansion=False)
7393:             print("\nâ\234\223 Intelligence layer works with legacy signal pack")
7394:         except Exception as e:
7395:             pytest.fail(f"Failed with legacy pack: {e}")
7396:
7397:         # Test with modern (full metadata)
7398:         modern_pack = ModernSignalPack()
7399:         try:
7400:             enriched_modern = create_enriched_signal_pack(modern_pack, enable_semantic_expansion=False)
7401:             print("â\234\223 Intelligence layer works with modern signal pack")
7402:         except Exception as e:
7403:             pytest.fail(f"Failed with modern pack: {e}")
7404:
7405:
7406: if __name__ == "__main__":
7407:     pytest.main([__file__, "-v", "-s"])
7408:
7409:
7410:
7411: ================================================================================
7412: FILE: tests/models/__init__.py
7413: ================================================================================
7414:
7415:
7416:
7417:
7418: ================================================================================
7419: FILE: tests/models/test_execution_plan.py
7420: ================================================================================
7421:
7422: """
7423: Unit tests for execution plan immutable data models.
7424: """
7425:
7426: import time
7427: from dataclasses import FrozenInstanceError
7428:
7429: import pytest
7430:
7431: from farfan_core.farfan_core.models.execution_plan import (
7432:     REQUIRED_TASK_COUNT,
7433:     ExecutableTask,
7434:     ExecutionPlan,
7435: )
7436:
7437: SHA256_HEX_LENGTH = 64
7438:
7439:
7440: class TestExecutableTaskImmutability:
7441:     """Test suite for ExecutableTask immutability guarantees."""
7442:
7443:     def test_executable_task_immutability(self) -> None:
7444:         """Verify FrozenInstanceError on modification attempts."""
7445:         task = ExecutableTask(
```

```
            task_id="T001",
            micro_question_context="Does policy address climate change?",
            target_chunk="chunk_001",
            applicable_patterns=("pattern_A", "pattern_B"),
            resolved_signals=("signal_X", "signal_Y"),
            creation_timestamp=time.time(),
            synchronizer_version="v1.0.0",
        )

        with pytest.raises(FrozenInstanceError):
            task.task_id = "T002"  # type: ignore[misc]

        with pytest.raises(FrozenInstanceError):
            task.micro_question_context = "Modified context"  # type: ignore[misc]

        with pytest.raises(FrozenInstanceError):
            task.target_chunk = "chunk_002"  # type: ignore[misc]

        with pytest.raises(FrozenInstanceError):
            task.creation_timestamp = time.time()  # type: ignore[misc]

    def test_tuple_fields_are_immutable(self) -> None:
        """Verify tuple fields cannot be modified."""
        task = ExecutableTask(
            task_id="T001",
            micro_question_context="Test context",
            target_chunk="chunk_001",
            applicable_patterns=("pattern_A", "pattern_B"),
            resolved_signals=("signal_X",),
            creation_timestamp=time.time(),
            synchronizer_version="v1.0.0",
        )

        with pytest.raises((AttributeError, TypeError)):
            task.applicable_patterns[0] = "modified"  # type: ignore[index]

        with pytest.raises((AttributeError, TypeError)):
            task.resolved_signals.append("signal_Z")  # type: ignore[attr-defined]


class TestExecutionPlanTaskCount:
    """Test suite for 300-task enforcement."""

    def test_execution_plan_rejects_wrong_task_count(self) -> None:
        """Verify ValueError for non-300 task counts."""
        tasks_299 = tuple(
            ExecutableTask(
                task_id=f"T{i:03d}",
                micro_question_context=f"Question {i}",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=time.time(),
                synchronizer_version="v1.0.0",
            )
            for i in range(299)
```

```
        )

        with pytest.raises(ValueError, match="requires exactly 300 tasks, got 299"):
            ExecutionPlan(plan_id="PLAN_299", tasks=tasks_299)

        tasks_301 = tuple(
            ExecutableTask(
                task_id=f"T{i:03d}",
                micro_question_context=f"Question {i}",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=time.time(),
                synchronizer_version="v1.0.0",
            )
            for i in range(301)
        )

        with pytest.raises(ValueError, match="requires exactly 300 tasks, got 301"):
            ExecutionPlan(plan_id="PLAN_301", tasks=tasks_301)

    def test_execution_plan_accepts_exactly_300_tasks(self) -> None:
        """Verify acceptance of exactly 300 tasks."""
        tasks_300 = tuple(
            ExecutableTask(
                task_id=f"T{i:03d}",
                micro_question_context=f"Question {i}",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=time.time(),
                synchronizer_version="v1.0.0",
            )
            for i in range(300)
        )

        plan = ExecutionPlan(plan_id="PLAN_300", tasks=tasks_300)
        assert len(plan.tasks) == REQUIRED_TASK_COUNT
        assert plan.plan_id == "PLAN_300"


class TestExecutionPlanDuplicateDetection:
    """Test suite for duplicate task_id detection."""

    def test_execution_plan_rejects_duplicate_task_ids(self) -> None:
        """Verify ValueError for duplicate task_ids."""
        tasks_with_duplicates = []
        for i in range(298):
            tasks_with_duplicates.append(
                ExecutableTask(
                    task_id=f"T{i:03d}",
                    micro_question_context=f"Question {i}",
                    target_chunk=f"chunk_{i:03d}",
                    applicable_patterns=("pattern_A",),
                    resolved_signals=("signal_X",),
                    creation_timestamp=time.time(),
```

```
7558:                     synchronizer_version="v1.0.0",
7559:                 )
7560:             )
7561:
7562:         tasks_with_duplicates.append(
7563:             ExecutableTask(
7564:                 task_id="T000",
7565:                 micro_question_context="Duplicate question",
7566:                 target_chunk="chunk_dup1",
7567:                 applicable_patterns=("pattern_B",),
7568:                 resolved_signals=("signal_Y",),
7569:                 creation_timestamp=time.time(),
7570:                 synchronizer_version="v1.0.0",
7571:             )
7572:         )
7573:
7574:         tasks_with_duplicates.append(
7575:             ExecutableTask(
7576:                 task_id="T001",
7577:                 micro_question_context="Another duplicate",
7578:                 target_chunk="chunk_dup2",
7579:                 applicable_patterns=("pattern_C",),
7580:                 resolved_signals=("signal_Z",),
7581:                 creation_timestamp=time.time(),
7582:                 synchronizer_version="v1.0.0",
7583:             )
7584:         )
7585:
7586:         with pytest.raises(
7587:             ValueError, match=r"contains duplicate task_ids.*\['T000', 'T001'\]"
7588:         ):
7589:             ExecutionPlan(plan_id="PLAN_DUP", tasks=tuple(tasks_with_duplicates))
7590:
7591:     def test_execution_plan_accepts_unique_task_ids(self) -> None:
7592:         """Verify acceptance of all unique task_ids."""
7593:         tasks_unique = tuple(
7594:             ExecutableTask(
7595:                 task_id=f"T{i:03d}",
7596:                 micro_question_context=f"Question {i}",
7597:                 target_chunk=f"chunk_{i:03d}",
7598:                 applicable_patterns=("pattern_A",),
7599:                 resolved_signals=("signal_X",),
7600:                 creation_timestamp=time.time(),
7601:                 synchronizer_version="v1.0.0",
7602:             )
7603:             for i in range(300)
7604:         )
7605:
7606:         plan = ExecutionPlan(plan_id="PLAN_UNIQUE", tasks=tasks_unique)
7607:         task_ids = {task.task_id for task in plan.tasks}
7608:         assert len(task_ids) == REQUIRED_TASK_COUNT
7609:
7610:
7611: class TestExecutionPlanIntegrityHash:
7612:     """Test suite for integrity hash computation."""
7613:
```

```
7614:     def test_execution_plan_integrity_hash_is_deterministic(self) -> None:
7615:         """Verify hash determinism across multiple computations."""
7616:         tasks = tuple(
7617:             ExecutableTask(
7618:                 task_id=f"T{i:03d}",
7619:                 micro_question_context=f"Question {i}",
7620:                 target_chunk=f"chunk_{i:03d}",
7621:                 applicable_patterns=("pattern_A", "pattern_B"),
7622:                 resolved_signals=("signal_X", "signal_Y"),
7623:                 creation_timestamp=1234567890.0 + i,
7624:                 synchronizer_version="v1.0.0",
7625:             )
7626:             for i in range(300)
7627:         )
7628:
7629:         plan = ExecutionPlan(plan_id="PLAN_HASH", tasks=tasks)
7630:
7631:         hash1 = plan.compute_integrity_hash()
7632:         hash2 = plan.compute_integrity_hash()
7633:         hash3 = plan.compute_integrity_hash()
7634:
7635:         assert hash1 == hash2 == hash3
7636:         assert len(hash1) == SHA256_HEX_LENGTH
7637:         assert all(c in "0123456789abcdef" for c in hash1)
7638:
7639:     def test_different_task_order_produces_different_hash(self) -> None:
7640:         """Verify hash changes with task ordering."""
7641:         base_tasks = [
7642:             ExecutableTask(
7643:                 task_id=f"T{i:03d}",
7644:                 micro_question_context=f"Question {i}",
7645:                 target_chunk=f"chunk_{i:03d}",
7646:                 applicable_patterns=("pattern_A",),
7647:                 resolved_signals=("signal_X",),
7648:                 creation_timestamp=1234567890.0,
7649:                 synchronizer_version="v1.0.0",
7650:             )
7651:             for i in range(300)
7652:         ]
7653:
7654:         plan1 = ExecutionPlan(plan_id="PLAN_ORDER1", tasks=tuple(base_tasks))
7655:         plan2 = ExecutionPlan(
7656:             plan_id="PLAN_ORDER2",
7657:             tasks=tuple(reversed(base_tasks)),
7658:         )
7659:
7660:         hash1 = plan1.compute_integrity_hash()
7661:         hash2 = plan2.compute_integrity_hash()
7662:
7663:         assert hash1 != hash2
7664:
7665:     def test_different_task_content_produces_different_hash(self) -> None:
7666:         """Verify hash changes with task content modifications."""
7667:         tasks1 = tuple(
7668:             ExecutableTask(
7669:                 task_id=f"T{i:03d}",
```

```
                micro_question_context="Context A",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=1234567890.0,
                synchronizer_version="v1.0.0",
            )
            for i in range(300)
        )

        tasks2 = tuple(
            ExecutableTask(
                task_id=f"T{i:03d}",
                micro_question_context="Context B",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=1234567890.0,
                synchronizer_version="v1.0.0",
            )
            for i in range(300)
        )

        plan1 = ExecutionPlan(plan_id="PLAN_CONTENT1", tasks=tasks1)
        plan2 = ExecutionPlan(plan_id="PLAN_CONTENT2", tasks=tasks2)

        hash1 = plan1.compute_integrity_hash()
        hash2 = plan2.compute_integrity_hash()

        assert hash1 != hash2

    def test_hash_is_sha256_format(self) -> None:
        """Verify hash conforms to SHA256 format."""
        tasks = tuple(
            ExecutableTask(
                task_id=f"T{i:03d}",
                micro_question_context=f"Question {i}",
                target_chunk=f"chunk_{i:03d}",
                applicable_patterns=("pattern_A",),
                resolved_signals=("signal_X",),
                creation_timestamp=time.time(),
                synchronizer_version="v1.0.0",
            )
            for i in range(300)
        )

        plan = ExecutionPlan(plan_id="PLAN_SHA256", tasks=tasks)
        integrity_hash = plan.compute_integrity_hash()

        assert isinstance(integrity_hash, str)
        assert len(integrity_hash) == SHA256_HEX_LENGTH
        int(integrity_hash, 16)


================================================================================
```

```
7726: FILE: tests/phases/__init__.py
7727: ================================================================================
7728:
7729: """Phase 0-11 Test Suite
7730:
7731: Tests for canonical phase contracts, boundaries, BLAKE3 hash stability,
7732: manifest completeness, failure propagation, and provenance verification.
7733: """
7734:
7735:
```